

MANUAL DE PROGRAMACIÓN

Índice

Introducción

- Propósito del Manual
- Propósito del proyecto
- Público Objetivo

Requisitos del sistema

- Hardware Necesario
- Software Requerido
- Dependencias y Librerías

Arquitectura del Sistema

- Diagrama de Arquitectura
- Descripción de Componentes

Estructura y desarrollo del Código

- WEB
- ALGORITMO
- ESP32
- ROBODK

Conexiones y Comunicaciones

- Configuración y conexión de la API REST
 - backend
 - frontend
- Configuración y conexión MQTT
 - WEB
 - ALGORITMO
 - ESP32
 - ROBODK

Introducción

Propósito del manual

Este documento detalla todos los aspectos técnicos relacionados con el sistema de información en cuestión. Su objetivo principal es familiarizar al personal técnico especializado en tareas críticas, que van desde el mantenimiento rutinario hasta la resolución de problemas complejos, así como la instalación y configuración inicial del sistema.

Este manual ofrece una guía completa que cubre desde los componentes físicos del sistema hasta su arquitectura de software. También se exploran los protocolos de seguridad implementados, las metodologías de respaldo de datos y otros aspectos relevantes que afectan el funcionamiento óptimo del sistema.

Además, se abordan los procedimientos de diagnóstico y solución de problemas, proporcionando al personal técnico las herramientas y conocimientos necesarios para enfrentar cualquier eventualidad durante el ciclo de vida del sistema. Se ofrecen pautas claras y procedimientos paso a paso para asegurar una ejecución efectiva de las tareas asignadas.

Este documento sirve como referencia esencial para el equipo técnico y como herramienta de capacitación para nuevos miembros del equipo. A través de ejemplos prácticos y casos de uso específicos, facilita la comprensión y el dominio de los diferentes aspectos del sistema de información, promoviendo una mayor eficiencia y efectividad en todas las operaciones relacionadas.

El manual actúa como una guía integral para una visión detallada y completa del sistema de información, equipando al personal técnico con los conocimientos y recursos necesarios para asegurar su funcionamiento fluido y confiable.

Propósito del manual

El propósito de nuestro proyecto de almacenes automatizados es optimizar la eficiencia operativa y la precisión en la gestión de inventarios mediante la implementación de tecnologías avanzadas. Se busca reducir significativamente los tiempos de procesamiento y los errores humanos, mejorando la velocidad y exactitud del picking y packing de productos. La automatización permite maximizar el uso del espacio disponible y optimizar las rutas de recolección, lo que conduce a una gestión más eficiente del almacén. Además, se pretende disminuir los costos operativos a largo plazo, aumentar la capacidad de respuesta ante la demanda del mercado y mejorar la seguridad en el entorno laboral. En conjunto, estos objetivos contribuyen a elevar la competitividad y rentabilidad de la empresa.

Público Objetivo

Este manual está diseñado para ser una guía integral para todo el personal técnico que participe en la implementación, comprensión y mantenimiento del sistema de información. Su objetivo es comprender los procedimientos para la instalación inicial y la resolución de problemas que surjan en los componentes de hardware y software.

Requisitos del sistema

Hardware necesario

A continuación, os adjuntamos una lista con el Hardware necesario para implementar la simulación de nuestro proyecto:

- ESP32-S3 (x2)
- Pc o portátil (windows | Mac | linux) con conexión a internet

Cabe recalcar, que en el caso de implementar el proyecto de cara al mundo real necesitaríamos los siguientes componentes que están comprendidos dentro de la simulación RoboDK:

(Robots)

- Robot ABB IRB 6640-130/3.2
- Robot UR20
- OMRON HD-1500 (AGV)

(Actuadores)

- Botón de emergencia
- Panel LED

(Sensores)

- LDR
- Ultrasonido
- Sensor de Barrera

Software necesario

De la parte de Software, para poder utilizar e instalar el proyecto de forma local en el ordenador necesitas lo siguiente:

- Navegador Web
- Visual Studio Code
- App de PostgreSQL
- Broker MQTT (recomendamos el uso de UI “MQTTX”)
- App de RoboDK
- App de Arduino
- Git/GitHub (opcional)

(Dependencias y librerías)

Para el diseño de la web hemos utilizado las siguientes librerías:

- <https://cdn.jsdelivr.net/npm/swiper/swiper-bundle.min.css> : librería css para hacer un deslizador de información

- <https://kit.fontawesome.com/3b8572695f.js> : Librería para la fuente de texto pertinente

- <https://cdnjs.cloudflare.com/ajax/libs/mqtt/5.5.5/mqtt.min.js> : Librería para poder utilizar las dependencias de mqtt en la web

Para la parte de Backend necesitamos las siguientes librerías y dependencias:

- Npm (node package manager)
 - Express (para crear un servidor)
 - Morgan (para ver las peticiones que llegan al servidor)
 - CORS (para tener los permisos de red necesarios para conectar la API con la web)
 - Pg (librería de PostgreSQL)

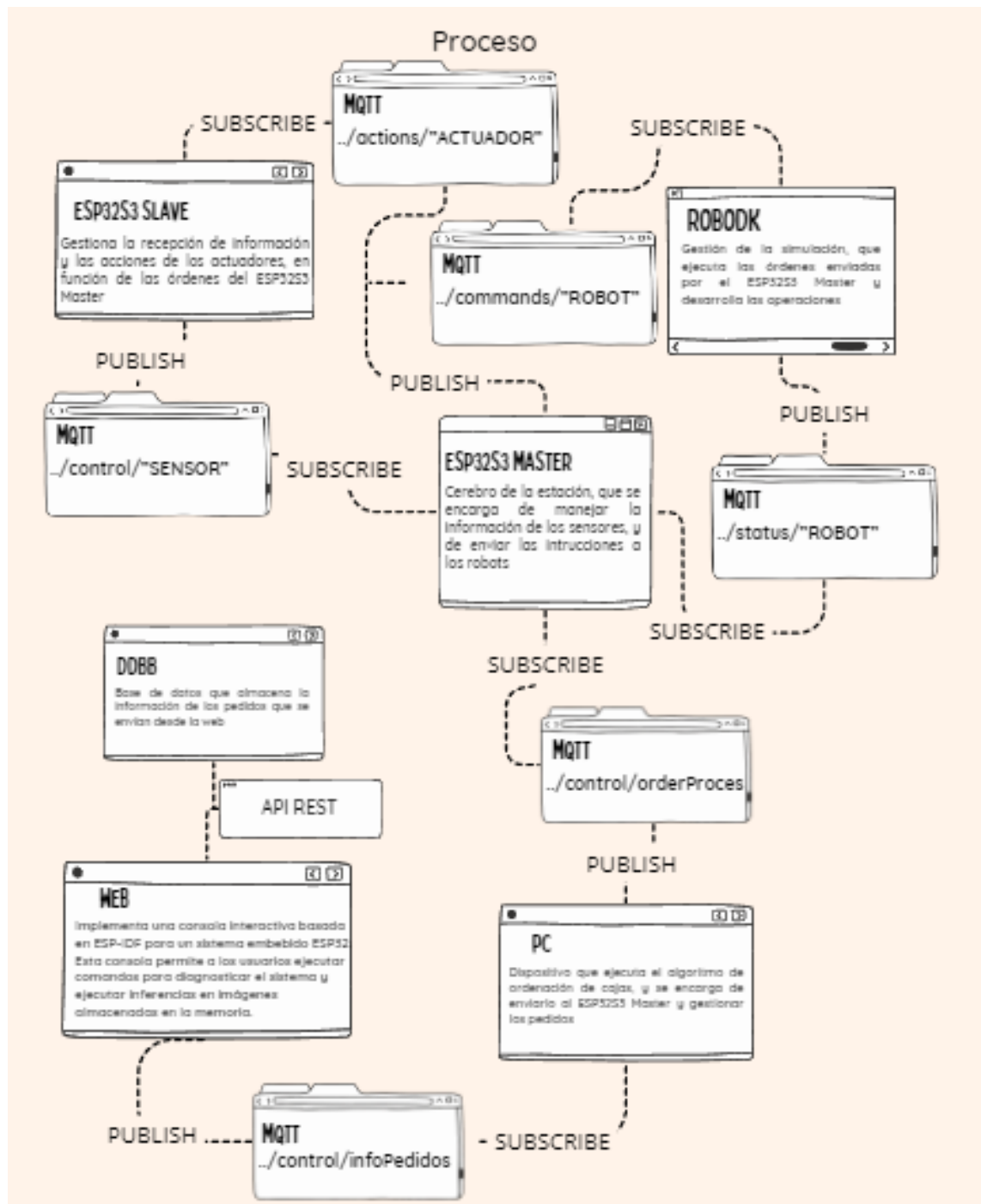
Robodk: json, paho.mqtt.client

PC: paho.mqtt.cpp,

ESP32S3: ArduinoJson .h, <LiquidCrystal_I2C.h>, <PubSubClient.h>,

Arquitectura del Sistema

Diagrama de arquitectura



Descripción de componentes

La arquitectura del proyecto automatizado se basa en una serie de componentes integrados que trabajan en conjunto para garantizar la eficiencia y precisión de las operaciones. A continuación, se describe la función de cada uno de estos componentes:

1. ESP32S3 Master y Slave

- **ESP32S3 Master:** Actúa como el cerebro principal de la estación. Su función es gestionar la información de los sensores y enviar las instrucciones necesarias a los robots para la ejecución de las tareas programadas. Este dispositivo también se encarga de procesar las órdenes recibidas desde otros componentes del sistema.
- **ESP32S3 Slave:** Complementa al Master gestionando la recepción de información y las acciones de los actuadores según las órdenes del Master. Esto asegura una distribución eficiente de las tareas y un control preciso de los dispositivos conectados.

2. Web

La interfaz web permite a los usuarios interactuar con el sistema, gestionando pedidos y controlando las operaciones. Los pedidos ingresados a través de la web son enviados a la base de datos para su procesamiento posterior.

3. RoboDK

RoboDK es el software de simulación que ejecuta las órdenes enviadas por el ESP32S3 Master. Se encarga de desarrollar las operaciones simuladas, garantizando que los movimientos y acciones de los robots sean precisos y estén optimizados para las tareas asignadas.

4. Base de Datos (DDBB)

La base de datos almacena toda la información relacionada con los pedidos recibidos desde la web. Esta información es esencial para el seguimiento y gestión de los pedidos, permitiendo una trazabilidad completa desde el ingreso del pedido hasta su finalización.

5. PC

El PC ejecuta el algoritmo de ordenación de cajas, un proceso crítico para la organización eficiente del almacén. Este dispositivo también se encarga de enviar los resultados del algoritmo al ESP32S3 Master y de gestionar los pedidos, asegurando que se cumplan las órdenes de manera eficiente.

6. API REST

La API REST facilita la comunicación entre diferentes partes del sistema, permitiendo el intercambio de información de manera eficiente. Esta interfaz se utiliza para gestionar la información de los pedidos y coordinar las acciones entre los diferentes componentes del sistema.

7. MQTT (Message Queuing Telemetry Transport)

El protocolo MQTT es crucial para la comunicación en el sistema. Permite publicar y suscribir mensajes entre los diferentes dispositivos y componentes, asegurando una comunicación fluida y en tiempo real. Las principales rutas de comunicación incluyen:

- **../control/infoPedidos:** Gestiona la recepción y procesamiento de la información de los pedidos.
- **../control/orderProcess:** Coordina el proceso de órdenes.
- **../actions/"ACTUADOR":** Maneja las acciones de los actuadores.
- **../commands/"ROBOT":** Envía comandos a los robots.
- **../control/"SENSOR":** Supervisa y controla los sensores.
- **../status/"ROBOT":** Publica y supervisa el estado de los robots.

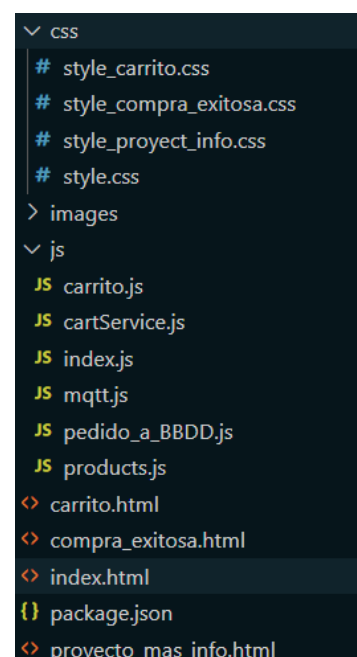
Esta arquitectura modular y bien definida garantiza que todos los componentes trabajen de manera cohesiva, permitiendo una operación eficiente y escalable del sistema automatizado. La integración de estos elementos permite gestionar de manera efectiva las tareas, optimizando el rendimiento y la productividad del proceso automatizado.

Estructura y desarrollo del código

WEB

Para la creación de la Web, hemos empleado el uso de 3 tecnologías: HTML (Estructura y contenido de la web), CSS (Diseño y presentación visual) y JavaScript (Interactividad y funcionalidad dinámica en la web.) que, en su conjunto, proporcionan al usuario con una experiencia atractiva, intuitiva y funcional de la web.

La estructura completa de la web en Visual Studio Code se refleja de la siguiente manera:



Primero tenemos el archivo **index.html**, que contiene toda la información respecto a la página principal de la web en donde tenemos 5 secciones diferentes:

Inicio: Presenta el título del proyecto y un resumen de la implementación de la automatización en los almacenes, incluyendo el uso de robots autónomos y diversas tecnologías.

Proyecto: Detalla la automatización de la gestión de pedidos y entregas, utilizando tecnologías como RoboDK, MQTT, y ESP32. Menciona la creación de una plataforma web interactiva para realizar pedidos de manera eficiente.

Tecnologías Utilizadas: Encontramos un listado de todas las tecnologías que se han utilizado para este proyecto.

Conócenos: En esta sección se presentan a todos los desarrolladores del proyecto.

Experimenta: Sección donde encontramos el sistema de pruebas de compra para ejecutar los pedidos en tiempo real.

```
<section id="proyecto-info">
  <h2>Información del Proyecto</h2>
  <p>Detalles sobre nuestro proyecto.</p>
</section>

<section id="tecnologias-utilizadas">
  <h2>Tecnologías Utilizadas</h2>
  <p>Descripción de las tecnologías que hemos usado.</p>
</section>

<section id="equipo">
  <h2>Equipo</h2>
  <p>Conoce a nuestro equipo de trabajo.</p>
</section>

<section id="pedido">
  <h2>Pedido</h2>
  <p>Haz tu pedido aquí.</p>
</section>

<footer>
  <p>&copy; 2024 Almacenes_NN. Todos los derechos reservados.</p>
</footer>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Almacenes_NN</title>
</head>
<body>
  <header class="header">
    <h1>Bienvenido a Almacenes_NN</h1>
  </header>

  <section id="inicio">
    <h2>Inicio</h2>
    <p>Bienvenido a nuestra página de inicio.</p>
  </section>
```

Una vez que se realice un pedido, la web te lleva a la página **carrito.html**.

A diferencia del **index.html** al tener la posibilidad de modificar la cantidad de paquetes y de poder eliminarlos, necesitamos del uso de JavaScript para la escritura y eliminación de elementos HTML en tiempo real de la web. Para ello, hacemos uso de los archivos **carrito.js**, **cartService.js** y **products.js**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Almacenes_NN</title>
</head>
<body>
  <header class="header">
    <h1>Almacenes_NN</h1>
  </header>

  <main>
    <section id="productos-container">
      <h2>Productos</h2>
      <!-- contenido de productos aquí -->
    </section>

    <section id="botones-y-totales">
      <h2>Botones y Totales</h2>
      <!-- Botones y totales aquí -->
    </section>

    <section id="mensaje-carrito-vacio">
      <h2>Mensaje Carrito Vacío</h2>
      <!-- Mensaje de carrito vacío aquí -->
    </section>
  </main>
```


Por último, tenemos el archivo **compra_exitosa.html** que se muestra una vez se realiza un pedido. Este archivo es bastante simple y sólo contiene 3 bloques principales que incluyen los 3 textos.

```
<body>
<header class="header">
<div class="menu">
  <a href="index.html" class="logo">
    
  </a>
</div>
</header>
```

```
<main>
<h1 id="lol">GRACIAS POR COMPRAR EN ALMACENES NN</h1>
<h2>AHORA MIRA COMO EL ALMACÉN AUTÓNOMO INTERACTÚA CON TU PEDIDO</h2>

<button class="boton_volver" onclick="location.href='index.html'">
  <p class="text"><strong>Volver a la página principal</strong></p>
</button>
</main>

<footer class="footer">
<div class="footer-text">
  <p>© 2024 NN - Todos los derechos reservados</p>
</div>
</footer>
</body>
</html>
```

Aparte, tenemos incluidos dos archivos más al proyecto: **fallo_conexion.html** y **proyecto_mas_info.html**.

*NOTA -> No hemos mostrado el código de los archivos css y js, ya que no veíamos oportuno mostrar muchas líneas de código que no son fáciles de entender y hemos optado por una visión más general del proyecto para entender la estructura por lo menos de la fachada.

Algoritmo

En esta sección del manual de programación, se detalla el funcionamiento del algoritmo central del dispositivo IoT, centrándose en la lógica interna y la composición de las clases utilizadas. Se explicarán las funcionalidades implementadas, cómo interactúan entre sí los distintos componentes del sistema y cómo se gestionan las tareas principales. Este análisis proporciona una comprensión profunda de la estructura del código y los mecanismos subyacentes que permiten el correcto desempeño del dispositivo.

Comenzamos con la explicación de las clases:

Target: La clase target es la encargada de guardar los puntos también llamados targets, que se utilizarán en RoboDK. Cabe resaltar que este archivo solo se encarga de la creación de la interfaz de la clase para el posterior almacenamiento de dichos puntos, pero no se encarga de generar estos puntos, ya que se calculan a partir del posicionamiento de las cajas (información que no se conoce en dicho momento).

```
cpp

#ifndef TARGET_H
#define TARGET_H

class target
{
private:
    float position_x;
    float position_y;
    float position_z;
    bool is_rotate;

public:
    target();
    target(float x, float y, float z, bool rotate);
    float get_position_x();
    float get_position_y();
    float get_position_z();
    bool get_is_rotate();
    void show_target();
};

#endif
```

Shipment: La clase shipment se encarga de almacenar los contenedores que se utilizan para un pedido, una vez se calcula la forma óptima de almacenar las cajas en los contenedores. Esta información se comunica mediante ficheros de texto/mqtt a RoboDK para su simulación.

```
cpp

#ifndef SHIPMENT_H
#define SHIPMENT_H

#include <vector>
#include "box.h"
#include "container.h"

class shipment
{
private:
    std::vector<box> V_boxes;
    std::vector<container> V_containers;

public:
    shipment();
    shipment(std::vector<box> V, std::vector<container> C);
    void show_shipment();
    int get_num_containers();
    int get_num_boxes();
    void save_shipment();
};

#endif
```

Order: La clase order sirve para almacenar la información de la orden que se produce cuando un cliente realiza un pedido a través de la web. Esto incluye las cajas con los componentes del pedido.

```
cpp

#ifndef ORDER_H
#define ORDER_H

#include <vector>
#include "box.h"

class order : public box
{
private:
    int num_boxes;
    std::vector<box> V_boxes_order;

public:
    order();
    void show_order();
    int get_num_boxes();
    std::vector<box> get_V_boxes();
    void save_order();
};

#endif
```

Box: La clase box proporciona todos los elementos necesarios para simular una caja en el código. Además, incluye todos los métodos esenciales para manipular y operar con estas cajas. Entre estos métodos se encuentran funciones para guardar las cajas en un archivo de texto o determinar si la caja se encuentra dentro de un contenedor, así como obtener información detallada sobre las dimensiones de la caja. Esta interfaz está diseñada para ser flexible y extensible, permitiendo una fácil integración con otras partes del sistema.

```
cpp

#ifndef BOX_H
#define BOX_H

#include <fstream>

class box
{
protected:
    char id_box;
    int length_box;
    int width_box;
    int height_box;
    int volume_box;
    bool placed;

public:
    box() : id_box(' '), length_box(0), width_box(0), height_box(0), volume_box(0), placed(0) {}
    box(int l, int w, int h, bool p, char id);
    int surface_area();
    char get_id();
    int get_length();
    int get_width();
    int get_height();
    int get_volume();
    void show_box();
    void set_placed(bool p);
    bool get_placed();
    void save_box(std::ofstream &file);
};

#endif
```

Container: La clase container tiene una interfaz parecida a la del archivo “box” a la hora de guardar las dimensiones, aunque difieren en los métodos, ya que cada archivo tiene una utilidad diferente. El archivo “container” contiene una matriz 3D que se utiliza para saber las posiciones de las cajas en un contenedor, para ello se utilizan números para identificar esta caja. Esta matriz es fundamental, ya que es la principal herramienta para determinar la posición de las siguientes cajas. Además, los contenedores tienen la opción de cambiar su tamaño para ajustarse de forma dinámica a la forma óptima para almacenar las cajas.

```
cpp
#ifndef CONTAINER_H
#define CONTAINER_H

#include <vector>
#include <fstream>
#include "box.h"

class container
{
protected:
    int length_container;
    int width_container;
    int height_container;
    std::vector<box> v_box;

public:
    int ***objects;
    container();
    container(int l, int w, int h);
    void add_box(box b);
    void show_container();
    void preset_objects();
    void set_objects(int ***objects_in);
    int get_length();
    int get_width();
    int get_height();
    int get_volume();
    int get_num_boxes();
    std::vector<box> get_v();
    int volume();
    int get_surface_area();
    void show_objects_container();
    int add_volume_boxes();
    void save_container(std::ofstream &file);
};

#endif
```

Algorithm: La clase Algorithm destaca como el componente más crucial dentro del repositorio, ya que integra todos los archivos precedentes para llevar a cabo los cálculos necesarios con el fin de determinar la disposición óptima de las cajas dentro de los contenedores, maximizando así la eficiencia del espacio disponible. Para ello, se hace uso de la matriz de contenedores mencionada anteriormente. Una vez establecidas las posiciones de cada caja, se generarán las coordenadas de los objetivos correspondientes, las cuales serán almacenadas en un archivo de texto o transmitidas a través de MQTT. Con toda esta información, se elaborará un envío hacia RoboDK.

```
cpp
#include "box.h"
#include "container.h"
#include "target.h"

#ifndef ALGORITHMS_H
#define ALGORITHMS_H

class Algorithms
{
public:
    std::vector<box> v_boxes_in_use;
    std::vector<container> v_containers_in_use;
    std::vector<target> v_targets;
    int length_container_in_use;
    int width_container_in_use;
    int height_container_in_use;
    int ***objects_in_use;
    int boxes_volume;

    Algorithms(int nb, int l, int w, int h, std::vector<box> V);
    void set_boxes(std::vector<box> V);
    void add_container(container in);
    void select_container();
    void place_boxes();
    bool prove_object(int ***objects, int w, int l, int h, box v, bool inverted = false);
    void order_boxes();
    void reset_objects();
    void erase_boxes_placed();
    void add_target(target in);
    void show_targets();
    void calculate_position_target(float *pos_x, float *pos_y, float *pos_h,
                                   int l, int w, int h,
                                   int o, int n, int m,
                                   bool inverted);
    void save_container(int n_container, int n_box);
    void save_results(target t_in, box b_in);
};

#endif
```

MAIN

En esta sección, se describe la función principal del programa, conocida como `main`. Esta función es el punto de entrada del programa y se encarga de inicializar los componentes esenciales, configurar las condiciones iniciales y gestionar el flujo principal de ejecución. A continuación, se detalla el funcionamiento de la función `main` y cómo coordina las diferentes partes del sistema para asegurar su correcto funcionamiento.

```
#define HEIGHT1 50
#define WIDTH1 100
#define LENGTH1 100

int main()
{
    int menu;
    for(;;)
    {
        // Espera hasta que exista el archivo "webOrder.txt"
        while (!std::filesystem::exists("D:\\repos\\Automatic_stock\\filesTXT\\webOrder.txt"))
            std::this_thread::sleep_for(std::chrono::seconds(1));

        // Creación de un objeto de la clase order
        order O1;

        // Creación de un objeto de la clase Algorithms
        // y llamada al constructor con los parámetros correspondientes
        Algorithms A1(O1.get_num_boxes(), LENGTH1, WIDTH1, HEIGHT1, O1.get_v_boxes());

        // Llamada al método place_boxes() del objeto A1
        A1.place_boxes();
    }
    return 0;
}
```

Inclusión de bibliotecas estándar y personalizadas:

- `#include <iostream>`: Incluye la biblioteca estándar de entrada y salida de C++.
- `#include <fstream>`: Incluye la biblioteca estándar de manipulación de archivos de C++.
- `#include <vector>`: Incluye la biblioteca estándar de vectores de C++.
- `#include <filesystem>`: Incluye la biblioteca estándar de manejo de archivos y directorios de C++.
- `#include <thread>`: Incluye la biblioteca estándar de manejo de hilos de C++ para realizar operaciones concurrentes.

- "src/headers/Algorithms.h", "src/headers/order.h",
"src/headers/shipment.h", "src/headers/box.h", "src/headers/container.h":

Incluye los archivos de encabezado (header files) necesarios para las clases y funciones utilizadas en el programa.

Definición de constantes:

- #define HEIGHT1 50: Define la altura del contenedor como 50.
- #define WIDTH1 100: Define el ancho del contenedor como 100.
- #define LENGTH1 100: Define la longitud del contenedor como 100.

Función principal main():

Bucle infinito for(;;):

- for(;;): Inicia un bucle infinito que se ejecutará continuamente hasta que el programa se cierre explícitamente con una instrucción return.

Espera de archivo:

while (!std::filesystem::exists("D:\\repos\\Automatic_stock\\filesTXT\\webOrder.txt")) {...}:

- Dentro del bucle infinito, se espera hasta que el archivo "webOrder.txt" exista en la ruta especificada ("D:\\repos\\Automatic_stock\\filesTXT\\"). Esto se realiza con la función std::filesystem::exists() de la biblioteca <filesystem>. Mientras el archivo no exista, el programa se bloqueará durante 1 segundo (std::this_thread::sleep_for(std::chrono::seconds(1));) antes de verificar nuevamente si el archivo existe.

Creación de objetos y ejecución de algoritmos:

- order O1;; Se crea un objeto de la clase order llamado O1.
- Algorithms A1(O1.get_num_boxes(), LENGTH1, WIDTH1, HEIGHT1, O1.get_V_boxes());: Se crea un objeto de la clase Algorithms llamado A1, pasando como parámetros el número de cajas, longitud, ancho y altura del contenedor, y el vector de cajas del objeto O1.
- A1.place_boxes();: Se llama al método place_boxes() del objeto A1, que realiza los algoritmos necesarios para colocar las cajas en el contenedor.

ALGORITMO:

En esta sección, se detallan las diversas funciones que componen el algoritmo del programa. Estas funciones son responsables de ejecutar tareas específicas, procesar datos y gestionar las operaciones necesarias para el correcto funcionamiento del sistema. A continuación, se explica cómo cada función contribuye al algoritmo general, proporcionando una visión detallada de su propósito y funcionamiento dentro del programa.

Constructor Algorithms::Algorithms(int nb, int l, int w, int h, std::vector<box> V):

En este constructor, se inicializan las dimensiones del contenedor y se crea un arreglo tridimensional `objects_in_use` para representar el espacio del contenedor. El constructor también reserva memoria para el vector de contenedores `V_containers_in_use` y asigna el vector de cajas `V` al vector `V_boxes_in_use`.

```
cpp Copiar código

Algorithms::Algorithms(int nb, int l, int w, int h, std::vector<box> V)
{
    length_container_in_use = l;
    width_container_in_use = w;
    height_container_in_use = h;
    objects_in_use = new int **[height_container_in_use];
    for (int i = 0; i < height_container_in_use; i++)
    {
        objects_in_use[i] = new int *[width_container_in_use];
        for (int j = 0; j < width_container_in_use; j++)
        {
            objects_in_use[i][j] = new int[length_container_in_use];
            for (int k = 0; k < length_container_in_use; k++)
            {
                objects_in_use[i][j][k] = 0;
            }
        }
    }

    V_containers_in_use.reserve(nb);
    V_boxes_in_use = V;
}
```

Método Algorithms::add_container(container in):

Este método agrega un contenedor al vector `V_containers_in_use`. Utiliza la función `push_back()` para agregar el contenedor `in` al final del vector.

```
cpp Copiar código

void Algorithms::add_container(container in)
{
    V_containers_in_use.push_back(in);
}
```

Método Algorithms::add_target(target in):

Similar al anterior, este método agrega un objetivo al vector `v_targets`. Utiliza la función `push_back()` para agregar el objetivo `in` al final del vector.

```
cpp Copiar código

void Algorithms::add_target(target in)
{
    v_targets.push_back(in);
}
```

Método Algorithms::prove_object(int ***objects, int w, int l, int h, box v, bool inverted):

Este método verifica si un objeto puede ser colocado en una posición específica del contenedor. Utiliza bucles `for` anidados para iterar sobre las dimensiones del contenedor y comprueba si alguna posición ya está ocupada por otro objeto.

```
cpp Copiar código

bool Algorithms::prove_object(int ***objects, int w, int l, int h, box v, bool inverted)
{
    int width, length, height = h + v.get_height();
    if(inverted)
    {
        width = w + v.get_length();
        length = l + v.get_width();
    } else
    {
        width = w + v.get_width();
        length = l + v.get_length();
    }

    for (int i = h; i < height; i++)
    {
        for (int j = w; j < width; j++)
        {
            for(int k = l; k < length; k++)
            {
                if (objects[i][j][k] != 0)
                {
                    return false;
                }
            }
        }
    }
    return true;
}
```

Método Algorithms::set_boxes(std::vector<box> V):

Este método establece el vector de cajas en uso. Asigna el vector `v` al vector `V_boxes_in_use`, que contiene las cajas disponibles para colocar en el contenedor.

```
cpp Copiar código

void Algorithms::set_boxes(std::vector<box> V)
{
    V_boxes_in_use = V;
}
```

Método `Algorithms::order_boxes()`:

Este método ordena las cajas en el vector `V_boxes_in_use` de mayor a menor volumen. Utiliza bucles `for` y una condición `while` para comparar el volumen de las cajas y reorganizarlas en orden descendente.

```
cpp Copiar código

void Algorithms::order_boxes()
{
    for (int i = 1; i < V_boxes_in_use.size(); i++)
    {
        int j = i - 1;
        int n = V_boxes_in_use[i].get_volume();
        while (j >= 0 && V_boxes_in_use[j].get_volume() < n)
        {
            box temp = V_boxes_in_use[j];
            V_boxes_in_use[j] = V_boxes_in_use[j + 1];
            V_boxes_in_use[j + 1] = temp;
            j--;
        }
    }
}
```

Método `Algorithms::show_targets()`:

Esta función muestra los objetivos almacenados en el vector `V_targets`. Itera sobre el vector y muestra cada objetivo llamando al método `show_target()` de la clase `target`.

```
cpp Copiar código

void Algorithms::show_targets()
{
    for (int i = 0; i < V_targets.size(); i++)
    {
        std::cout << "Target " << i + 1 << std::endl;
        V_targets[i].show_target();
    }
}
```


Método Algorithms::reset_objects():

Este método restablece los valores del arreglo tridimensional `objects_in_use` a cero. Recorre cada dimensión del arreglo y establece el valor en cero.

```
cpp Copiar código

void Algorithms::reset_objects()
{
    for (int i = 0; i < height_container_in_use; i++)
    {
        for (int j = 0; j < width_container_in_use; j++)
        {
            for (int k = 0; k < length_container_in_use; k++)
            {
                objects_in_use[i][j][k] = 0;
            }
        }
    }
}
```

Método Algorithms::erase_boxes_placed():

Este método elimina las cajas colocadas del vector `V_boxes_in_use`. Itera sobre el vector y elimina las cajas que tienen el indicador `placed` en true.


```
cpp Copiar código

void Algorithms::erase_boxes_placed()
{
    for (int i = 0; i < V_boxes_in_use.size(); i++)
    {
        if (V_boxes_in_use[i].get_placed())
        {
            V_boxes_in_use.erase(V_boxes_in_use.begin() + i);
            i--;
        }
    }
}
```

Método Algorithms::calculate_position_target(float *pos_x, float *pos_y, float *pos_h, int l, int w, int h, int o, int n, int m, bool inverted):

Esta función calcula la posición del objetivo teniendo en cuenta las coordenadas de la esquina superior izquierda del contenedor, el desplazamiento del objetivo y si está invertido. Calcula las coordenadas x, y y z del objetivo y las asigna a las direcciones de memoria proporcionadas.

cpp

 Copiar código

```


void Algorithms::calculate_position_target(float *pos_x, float *pos_y, float *pos_h,
                                          int l, int w, int h,
                                          int o, int n, int m,
                                          bool inverted)
{
    if(!inverted)
    {
        *pos_x = ((2.0 * (float)(l) + (float)(o)) / 2.0);
        *pos_y = ((2.0 * (float)(w) + (float)(n)) / 2.0);
        *pos_h = (float)(h + m);
    }
    else
    {
        *pos_x = ((2.0 * (float)(l) + (float)(n)) / 2.0);
        *pos_y = ((2.0 * (float)(w) + (float)(o)) / 2.0);
        *pos_h = (float)(h + m);
    }
}

```

Método Algorithms::save_container(int n_container, int n_box):

Este método guarda información sobre el contenedor y el número de cajas en un archivo de texto. Abre el archivo en modo de anexión y escribe la información del contenedor y el número de cajas.

cpp

 Copiar código

```

void Algorithms::save_container(int n_container, int n_box)
{
    std::ofstream file;
    file.open("D:\\repos\\Automatic_stock\\filesTXT\\results.txt", std::ios::app);
    file << "Container: " << n_container << std::endl;
    file << "Number-boxes: " << n_box << std::endl;
    file.close();
}

```

Método Algorithms::save_results(target t_in, box b_in):

Este método guarda información sobre la posición del objetivo y el ID de la caja asociada en un archivo de texto. Abre el archivo en modo de anexión y escribe la información del ID de la caja y la posición del objetivo en el formato especificado.

```

cpp Copiar código

void Algorithms::save_results(target t_in, box b_in)
{
    std::ofstream file;
    file.open("D:\\repos\\Automatic_stock\\filesTXT\\results.txt", std::ios::app);
    file << "ID_Box: " << b_in.get_id() << std::endl;
    if(t_in.get_is_rotate())
    {
        file << "Position: (" << t_in.get_position_x() << "," << t_in.get_position_y() <<
    } else
    {
        file << "Position: (" << t_in.get_position_x() << "," << t_in.get_position_y() <<
    }
    file.close();
}

```

Algoritmo de colocación de cajas:

Inicialización y ordenación de cajas: Se inicializan las variables necesarias para el seguimiento del proceso de colocación de las cajas, como el número de caja actual, el número de contenedor y algunas variables de posición. Luego, se ordenan las cajas de mayor a menor volumen para optimizar el proceso de colocación.

```

cpp Copiar código

int n_box = 1, n_container = 1;
int m, n, o, t = 0;
float pos_x, pos_y, pos_z;
order_boxes();
bool all_boxes_placed = false;

```

Bucle principal: El bucle principal controla el proceso de colocación de las cajas y se ejecuta hasta que todas las cajas hayan sido colocadas en los contenedores.

```


cpp Copiar código

while(!all_boxes_placed)

```

Verificación de caja no colocada: Se verifica si una caja específica aún no ha sido colocada en ningún contenedor.


cpp

 Copiar código

```
for (int i = 0; i < V_boxes_in_use.size(); i++)
{
    if(!V_boxes_in_use[i].get_placed())
    {
```

Bucles anidados para colocar cajas: Se utilizan bucles anidados para recorrer todas las posiciones posibles dentro del contenedor donde se podría colocar una caja. Se comprueba si una caja puede caber en una posición específica dentro del contenedor.


cpp

 Copiar código

```
for(int h = 0; h < height_container_in_use; h++)
{
    for (int w = 0; w < width_container_in_use; w++)
    {
        for (int l = 0; l < length_container_in_use; l++)
        {
            if (V_boxes_in_use[i].get_height() <= height_container_in_use - h
                && V_boxes_in_use[i].get_width() <= width_container_in_use - w
                && V_boxes_in_use[i].get_length() <= length_container_in_use - l
                && prove_object(objects_in_use, w, l, h, V_boxes_in_use[i]))
            {
```

Colocación de la caja: Si se encuentra una posición adecuada para colocar la caja, se actualiza la matriz de objetos dentro del contenedor para reflejar la presencia de la caja en esa posición. También se calcula la posición del objetivo asociado a esa caja y se marca la caja como colocada.

cpp

 Copiar código

```
for (m = 0; m < V_boxes_in_use[i].get_height(); m++)
{
    for (n = 0; n < V_boxes_in_use[i].get_width(); n++)
    {
        for (o = 0; o < V_boxes_in_use[i].get_length(); o++)
        {
            objects_in_use[h + m][w + n][l + o] = n_box;
        }
    }
}

calculate_position_target(&pos_x, &pos_y, &pos_z, l, w, h, o, n, m, false);
target T1(pos_x, pos_y, pos_z, false);
add_target(T1);
w = width_container_in_use;
l = length_container_in_use;
h = height_container_in_use;
n_box++;
V_boxes_in_use[i].set_placed(true);
```

Creación del contenedor y reseteo: Una vez se han colocado todas las cajas posibles en un contenedor, se crea un nuevo contenedor con las cajas colocadas y se añade al vector de contenedores en uso. Se guardan los resultados de la colocación de las cajas.

```
cpp Copiar código

container C1(width_container_in_use, length_container_in_use, height_container_in_use);
C1.set_objects(objects_in_use);

for(int i = 0; i < V_boxes_in_use.size(); i++)
{
    if(V_boxes_in_use[i].get_placed())
    {
        C1.add_box(V_boxes_in_use[i]);
    }
}
add_container(C1);
save_container(n_container, C1.get_num_boxes());
n_container++;
for(int i = 0; i < V_boxes_in_use.size(); i++)
{
    if(V_boxes_in_use[i].get_placed())
    {
        save_results(V_targets[t], V_boxes_in_use[i]);
        t++;
    }
}
reset_objects();
erase_boxes_placed();
```

Condición de salida del bucle principal: La condición de salida del bucle principal comprueba si todas las cajas han sido colocadas en los contenedores. Si todas las cajas han sido colocadas, se actualiza la bandera `all_boxes_placed` a `true` y se sale del bucle principal.

```
cpp Copiar código

if(V_boxes_in_use.empty())
{
    all_boxes_placed = true;
}
```

ESP32S3 MASTER

Archivo config.h:

El archivo config.h contiene la configuración del dispositivo ESP32-S3 Master, incluyendo parámetros de comunicación, configuración de red WiFi, configuración MQTT, definición de topics MQTT, y configuración de SSL. Aquí se describen las secciones principales de este archivo:

- **COMM BAUDS y MQTT_BUFFER_SIZE:** Define la velocidad de comunicación serie y el tamaño del búfer MQTT respectivamente.
- **LOGGER_ENABLED y LOG_LEVEL:** Configura el logger para la salida por consola serie y establece el nivel de log.
- **DEVICE:** Define el ID del dispositivo ESP32 y el ID del dispositivo GIIROB PR2.
- **WIFI:** Configura el nombre y la contraseña de la red WiFi a la que se conectará el dispositivo.
- **MQTT:** Configura la dirección IP y el puerto del servidor MQTT, así como el nombre de usuario y contraseña si es necesario.
- **SSL:** Permite la configuración de SSL/TLS para conexiones seguras si se define **SSL_ROOT_CA**.
- **Definición de topics MQTT:** Se definen los topics MQTT utilizados para diferentes propósitos como saludo, procesamiento de órdenes, control de acciones, control de sensores, entre otros.
- **SDA y SCL:** Define los pines SDA y SCL para la comunicación I2C.

Este archivo es crucial para configurar el dispositivo y establecer la comunicación adecuada con el servidor MQTT y otros dispositivos en la red.

Archivos de clases:

Los archivos de cabecera order.h, box.h y container.h contienen las definiciones de clases que representan diferentes entidades en el sistema. Aquí hay una descripción breve de cada uno:

- **order.h:** Este archivo describe la clase Order, que representa una orden o pedido en el sistema. La clase puede contener información como el ID del pedido, los productos solicitados, la cantidad de cada producto y cualquier otra información relevante asociada con el pedido.
- **box.h:** En este archivo se define la clase Box, que representa una caja en el sistema. La clase puede incluir atributos como el ID de la caja, su posición física en el almacén, el estado de la caja (llena/vacía), y cualquier otra información relacionada con la gestión de cajas.
- **container.h:** Este archivo contiene la definición de la clase Container, que representa un contenedor utilizado para el almacenamiento y transporte de productos en el sistema. La clase puede tener propiedades como el ID del contenedor, su capacidad, los productos almacenados en él y su estado (lleno/vacío).

Archivo f_funciones:

El archivo `f_funciones` contiene una serie de funciones que desempeñan roles específicos en el sistema de gestión del almacén. Estas funciones están diseñadas para realizar tareas variadas, desde el control de dispositivos hasta la manipulación de datos y la comunicación con otros sistemas. A continuación, se describen las funciones y su papel en el sistema de manera más detallada.

La función `getOrder` procesa mensajes de entrada, convirtiéndolos en objetos de tipo `order` que representan pedidos con contenedores y cajas. Este proceso implica la deserialización del mensaje JSON entrante, la construcción de un objeto `order` temporal con los datos obtenidos del mensaje, la creación de contenedores y cajas a partir de la estructura del mensaje JSON, y la adición de estos objetos al pedido. Una vez completado el procesamiento, la función devuelve el objeto `order` construido, listo para ser utilizado en la lógica del programa.

```
cpp Copiar código

order getOrder(String incomingMessage) {
    JsonDocument doc;
    DeserializationError err = deserializeJson(doc, incomingMessage);
    order tempOrder;
    if (!err) {
        tempOrder = order(doc["N_CONT"].as<int>(), doc["ID_PEDIDO"]);
        JsonArray CONTAINER = doc["CONTAINER"];
        for (JsonVariant v : CONTAINER) {
            container inputContainer = container(v["N_BOX"].as<int>(), v["ID_CONT"]);
            JsonArray Positions = v["BOXES"];
            for (JsonVariant w : Positions) {
                box inputBox = box(w["ID_BOX"], w["POSE"]);
                String id = w["ID_BOX"];
                inputContainer.add_box(inputBox);
            }
            tempOrder.add_container(inputContainer);
        }
        tempOrder.set_Idboxes();
    }
    return tempOrder;
}
```

Archivo g_comunicaciones:

El archivo "g_comunicaciones" sirve como el núcleo central donde se gestionan todas las acciones de control y comunicación del proceso. Aquí se definen las funciones y métodos que permiten la interacción con otros dispositivos y sistemas a través de protocolos de comunicación como MQTT, así como el procesamiento de los mensajes entrantes y salientes para coordinar las operaciones del sistema. Esencialmente, este archivo actúa como el mediador principal entre el dispositivo y su entorno, facilitando la transferencia de datos y la ejecución de acciones según las necesidades del sistema.

if (strcmp(topic, HELLO_TOPIC) == 0): Este bloque verifica si el mensaje ha sido recibido en el topic HELLO_TOPIC. Este topic generalmente se utiliza para establecer la conexión inicial entre dispositivos y verificar la disponibilidad de otros dispositivos en la red.

En este caso, el bloque está vacío, lo que indica que no se realizan acciones específicas cuando se recibe un mensaje en este topic.

if (strcmp(topic, STATION_STATUS_TOPIC) == 0): Este bloque verifica si el mensaje ha sido recibido en el topic STATION_STATUS_TOPIC. Este topic se utiliza para recibir actualizaciones sobre el estado de la estación de trabajo.

Dentro de este bloque, se deserializa el mensaje JSON recibido para extraer el estado de la estación.

Dependiendo del estado recibido ("ready", "start", "finished"), se ejecutan diferentes acciones. Por ejemplo, si la estación está lista ("ready"), se muestra un mensaje en una pantalla LCD. Si la estación comienza ("start"), se envían mensajes a otros dispositivos para iniciar procesos. Si la estación termina ("finished"), se envían mensajes para detener procesos.

if (strcmp(topic, DESPALETIZADO_STATUS_TOPIC) == 0): Este bloque verifica si el mensaje ha sido recibido en el topic DESPALETIZADO_STATUS_TOPIC. Este topic se utiliza para recibir actualizaciones sobre el estado del proceso de despaletizado.

Dentro de este bloque, se deserializa el mensaje JSON recibido para extraer el estado del proceso.

Si el estado indica que el despaletizado está listo ("ready"), se realizan acciones específicas, como enviar comandos para sacar cajas de un palet.

if (strcmp(topic, CONVEYOR1_STATUS_TOPIC) == 0): Este bloque verifica si el mensaje ha sido recibido en el topic CONVEYOR1_STATUS_TOPIC. Este topic se utiliza para recibir actualizaciones sobre el estado de una cinta transportadora.

Dentro de este bloque, se deserializa el mensaje JSON recibido para extraer el estado de la cinta transportadora.

Dependiendo del estado recibido ("ready", "object"), se envían comandos para iniciar o detener la cinta transportadora.

if (strcmp(topic, ASIGNACION_STATUS_TOPIC) == 0): Este bloque verifica si el mensaje ha sido recibido en el topic ASIGNACION_STATUS_TOPIC. Este topic se utiliza para recibir actualizaciones sobre el estado del proceso de asignación.

Dentro de este bloque, se deserializa el mensaje JSON recibido para extraer el estado del proceso.

Si el estado indica que el proceso de asignación ha comenzado ("start"), se realizan acciones específicas, como enviar comandos para colocar cajas en contenedores.

if (strcmp(topic, SENSOR1_STATUS_TOPIC) == 0): Este bloque verifica si el mensaje ha sido recibido en el topic SENSOR1_STATUS_TOPIC. Este topic se utiliza para recibir actualizaciones sobre el estado de un sensor.

Dentro de este bloque, se deserializa el mensaje JSON recibido para extraer el estado del sensor.

Si el sensor está listo ("ready"), se envían comandos para iniciar el sensor.

if (strcmp(topic, AGV_STATUS_TOPIC) == 0): Este bloque verifica si el mensaje ha sido recibido en el topic AGV_STATUS_TOPIC. Este topic se utiliza para recibir actualizaciones sobre el estado de un vehículo de guiado automático (AGV).

Dentro de este bloque, se deserializa el mensaje JSON recibido para extraer el estado del AGV.

Si el AGV está listo ("ready"), se envían comandos para iniciar el AGV.

if (strcmp(topic, ORDERPROCES_TOPIC) == 0): Este bloque verifica si el mensaje ha sido recibido en el topic ORDERPROCES_TOPIC. Este topic se utiliza para recibir órdenes de procesamiento.

Dentro de este bloque, se llama a la función getOrder() para procesar el mensaje JSON recibido y se establece el estado de la estación para comenzar el procesamiento.

ESP32S3 SLAVE

Archivo config.h:

El archivo config.h es un componente crucial en el sistema, ya que contiene todas las definiciones y configuraciones necesarias para los pines de los sensores, la configuración de la red WiFi, la configuración del protocolo MQTT, la definición de los tópicos MQTT, y las asignaciones de los pines de la cámara. Este archivo actúa como una cabecera centralizada que permite una fácil modificación y gestión de todas las configuraciones hardware y de red del dispositivo.

Archivo f_funciones:

El archivo f_funciones contiene la implementación de un buffer circular y la gestión de varios sensores que interactúan con topics MQTT. A continuación, se describe cada componente y su funcionalidad.

La clase Buffer_Circ es responsable de gestionar un buffer circular que almacena mensajes de tipo String. Este buffer se utiliza para manejar mensajes entrantes y salientes de manera eficiente.

```
cpp Copiar código

struct Buffer_Circ {
    String* data; // Puntero a array de Strings
    int bufIN, bufOUT, contador, capacidad;

    // Constructor para inicializar el buffer
    Buffer_Circ(int cap) : bufIN(0), bufOUT(0), contador(0), capacidad(cap) {
        data = new String[capacidad]; // Asigna memoria para el array de Strings
    }

    // Destructor para liberar memoria
    ~Buffer_Circ() {
        delete[] data; // Libera la memoria asignada
    }
};
```

El archivo define varias funciones para operar sobre el buffer circular:


get_item: Extrae y devuelve un mensaje del buffer. Si el buffer está vacío, devuelve una cadena vacía.

```
cpp Copiar código

String get_item(Buffer_Circ* buff)
{
    if (vacía(buff)) // Comprueba si el buffer está vacío
        return ""; // Si está vacío, devuelve una cadena vacía
    else
    {
        String item = buff->data[buff->bufOUT]; // Obtiene el elemento en bufOUT
        buff->contador--; // Decrementa el contador
        buff->bufOUT = (buff->bufOUT + 1) % buff->capacidad; // Actualiza bufOUT de manera circular
        return item; // Devuelve el elemento
    }
}
```

put_item: Inserta un mensaje en el buffer. Devuelve 0 si la inserción es exitosa, y -1 si el buffer está lleno.


cpp

 Copiar código

```
int put_item(const String& msg, Buffer_Circ* buff)
{
    if (llena(buff)) // Comprueba si el buffer está lleno
        return -1; // Si está lleno, devuelve -1
    else
    {
        buff->data[buff->bufIN] = msg; // Inserta el mensaje en bufIN
        buff->contador++; // Incrementa el contador
        buff->bufIN = (buff->bufIN + 1) % buff->capacidad; // Actualiza bufIN de manera circular
        return 0; // Devuelve 0 indicando éxito
    }
}
```

Vacia: Verifica si el buffer está vacío. Devuelve true si está vacío, de lo contrario false.


cpp

 Copiar código

```
bool vacia(Buffer_Circ* buff)
{
    return buff->contador == 0; // Devuelve true si el contador es 0
}
```

Llena: Verifica si el buffer está lleno. Devuelve true si está lleno, de lo contrario false.

cpp

 Copiar código

```
bool llena(Buffer_Circ* buff)
{
    return buff->contador == buff->capacidad; // Devuelve true si el contador es igual a la capacidad
}
```

listanddelete: Extrae todos los mensajes del buffer, los convierte en un formato JSON y los envía a un tópico MQTT específico.

```
cpp Copiar código

void listanddelete(Buffer_Circ* buff, String& result)
{
    StaticJsonDocument<1000> doc; // Crea un documento JSON con capacidad adecuada
    JsonArray data = doc.createNestedArray("data"); // Crea un array JSON anidado

    while (!vacía(buff)) // Mientras el buffer no esté vacío
    {
        data.add(buff->data[buff->bufOUT]); // Añade el elemento actual al array JSON
        buff->contador--; // Decrementa el contador
        buff->bufOUT = (buff->bufOUT + 1) % buff->capacidad; // Actualiza bufOUT de manera circular
    }
    serializeJson(doc, result); // Serializa el documento JSON en la cadena result

    enviarMensajePorTópico(INCIDENCE_TOPIC, result); // Envía el mensaje por el tópico MQTT
    delay(100); // Espera un breve tiempo
}
```

Tam: Devuelve el número actual de mensajes en el buffer.

```
cpp Copiar código

int tam(Buffer_Circ* buff)
{
    return buff->contador; // Devuelve el número de elementos en el buffer
}
```

Gestión de Sensores

El archivo también contiene funciones para manejar diferentes sensores y sus interacciones con tópicos MQTT:

Us: Gestiona un sensor de proximidad (ultrasonido). Mide la distancia y envía un mensaje si la distancia es menor que un umbral específico.

Inicialización del mensaje JSON: Se crea un objeto JSON para representar la lectura del sensor de proximidad. Se asigna un ID único al sensor ("us001") y se define el tipo de proximidad ("near"), indicando que el objeto está cerca del sensor.

```
cpp Copiar código

void us(void *pvParameters)
{
    while(1)
    {
        StaticJsonDocument<200> doc; // Crea el mensaje en formato JSON
        doc["ID_SENSOR"] = "us001"; // ID del sensor
        doc["PROXIMITY_TYPE"] = "near"; // Tipo de proximidad
        String state_json;
        serializeJson(doc, state_json); // Serializa el documento JSON en una cadena
    }
}
```

Envío de pulso ultrasónico: Se realiza la secuencia de envío de pulsos ultrasónicos al sensor. Esto implica activar brevemente el pin de disparo del sensor ultrasónico, esperar y luego desactivarlo. Esta secuencia permite al sensor iniciar la medición de distancia.

```
cpp Copiar código

digitalWrite(US_SENSOR1, LOW);
delay(200);

// Generar un pulso de 10 microsegundos en el pin de disparo
digitalWrite(US_SENSOR1, HIGH);
delay(100);
digitalWrite(US_SENSOR1, LOW);
```

Medición de la distancia: Se utiliza la función `pulseIn()` para medir la duración del eco del pulso ultrasónico. A partir de esta duración, se calcula la distancia del objeto al sensor utilizando la fórmula de velocidad del sonido.

```
cpp Copiar código

// Leer la duración del pulso en el pin de eco
long time = pulseIn(US_SENSOR2, HIGH);

// Calcular la distancia en cm
int distance = time * 0.034 / 2;
```

Envío del mensaje MQTT: Si la distancia medida es menor que un umbral específico (en este caso, 5 cm), se considera que hay un objeto cerca del sensor. Se serializa el objeto JSON en una cadena y se envía a un tópico MQTT dedicado (`PROXIMITY_TOPIC`) utilizando la función `enviarMensajePorTopic()`. Esto permite que otros dispositivos o sistemas reciban y actúen en consecuencia según la proximidad del objeto detectado.

```
if (distance < 5)
{
    enviarMensajePorTopic(PROXIMITY_TOPIC, state_json); // Envía el mensaje por el tópico
    while(distance < 5)
    {
        long time = pulseIn(US_SENSOR2, HIGH);
        int distance = time * 0.034 / 2;
        vTaskDelay(100 / portTICK_PERIOD_MS); // Espera un breve tiempo
    }
}
```

Espera antes de la próxima lectura: Después de completar una iteración.

```
cpp                                                                    Copiar código

    // Esperar un breve tiempo antes de tomar otra lectura
    vTaskDelay(200 / portTICK_PERIOD_MS);
}
}
```

Readbutton: Lee continuamente el estado de un botón de emergencia. Si el botón está pulsado, envía un mensaje de emergencia a un tópico MQTT.

Inicialización y lectura continua del estado del botón: El bucle `while` se ejecuta continuamente para monitorear el estado del botón de emergencia. Dentro del bucle, se crea un mensaje JSON que indica la activación de una emergencia. Si el botón está pulsado, se envía este mensaje por el tópico MQTT de emergencia. El bucle espera hasta que el botón se suelte antes de continuar.

```
cpp                                                                    Copiar código

void readbutton(void *pvParameters) // Lee el botón continuamente, si está pulsado envía
{
    try
    {
        while(1)
        {
            vTaskDelay(200 / portTICK_PERIOD_MS); // Espera un breve tiempo

            StaticJsonDocument<200> doc; // Crea el mensaje en formato JSON
            doc["ID_SENSOR"] = "emb001"; // ID del sensor
            doc["EMERGENCY_TYPE"] = "stop"; // Tipo de emergencia
            String state_json;
            serializeJson(doc, state_json); // Serializa el documento JSON en una cadena

            if(digitalRead(PIN_BUTTON) == LOW) // Si el botón está pulsado
            {
                enviarMensajePorTopic(EMERGENCY_TOPIC, state_json); // Envía el mensaje por el tópico
                while(digitalRead(PIN_BUTTON) == LOW)
                {
                    vTaskDelay(100 / portTICK_PERIOD_MS); // Espera un breve tiempo
                }
            }
        }
    }
}
```

Manejo de excepciones: La estructura try-catch maneja cualquier excepción que pueda ocurrir durante la ejecución de la función. Si se produce una excepción, se crea un mensaje JSON que indica el tipo de excepción y se serializa. Sin embargo, en el código proporcionado, no se especifica ninguna acción adicional después de la captura de la excepción.

```
catch(...)
{
    StaticJsonDocument<200> doc; // Crea el mensaje en formato JSON
    doc["ID_SENSOR"] = "EXCEPTION"; // ID del sensor
    doc["EXCEPTION_TYPE"] = "Error with sensor"; // Tipo de excepción
    String state_json;
    serializeJson(doc, state_json); // Serializa el documento JSON en una cadena
}
}
```

Lectura del sensor de luz y envío de mensajes de estado: En un bucle infinito, la función lee el valor analógico del sensor de luz (LDR) utilizando la función `analogRead()`. Luego, crea un mensaje JSON que contiene el ID del sensor y los datos recopilados del sensor. Este mensaje se serializa en formato JSON y se almacena en una cadena.

```
cpp Copiar código

void iluminacion(void *pvParameters)
{
    while(1)
    {
        int adcVal = analogRead(PIN_ANALOG_IN); // Lee el valor del sensor LDR

        StaticJsonDocument<200> doc; // Crea el mensaje en formato JSON
        doc["ID_SENSOR"] = "ldr001"; // ID del sensor
        doc["DATA"] = "lightoff"; // Datos del sensor
        String state_json;
        serializeJson(doc, state_json); // Serializa el documento JSON en una cadena
    }
}
```

Verificación del nivel de luz y envío de mensajes de estado: El valor leído del sensor se compara con un umbral predefinido (700 en este caso). Si el valor leído del sensor indica poca luz (es decir, si es mayor que el umbral), se envía un mensaje al tópico MQTT de iluminación indicando que la luz está apagada. El bucle espera hasta que el valor del sensor indique que hay suficiente luz antes de continuar.


```
if(adcVal > 700) // Si el LDR capta poca luz
{
    enviarMensajePorTopic(ILUMINATION_TOPIC, state_json); // Envía el mensaje por el tópico
    while(adcVal = analogRead(PIN_ANALOG_IN) > 700)
    {
        vTaskDelay(100/ portTICK_PERIOD_MS); // Espera un breve tiempo
    }
}
}
```

Archivo g_comunicaciones:

El archivo g_comunicaciones se encarga de gestionar la comunicación MQTT y el almacenamiento de mensajes en un buffer circular. Este archivo incluye la implementación de la suscripción a tópicos MQTT, el envío de mensajes a través de estos topic y la gestión de mensajes recibidos, almacenándolos en un buffer circular.

Buffer Circular: Se instancia un buffer circular llamado Buffer con una capacidad de 100 elementos. Este buffer se utiliza para almacenar mensajes recibidos de distintos tópicos MQTT.

cpp

 Copiar código

```
#include "Config.h"

Buffer_Circ Buffer(100); // Instancia del buffer con capacidad de 100 elementos
```

Funciones de Suscripción (suscribirseATopics()): Esta función se encarga de suscribirse a varios tópicos MQTT relevantes (HELLO_TOPIC, ILUMINATION_TOPIC, PROXIMITY_TOPIC, y EMERGENCY_TOPIC)


cpp

 Copiar código

```
void suscribirseATopics()
{
    // Añadir suscripciones a los topics MQTT
    mqtt_subscribe(HELLO_TOPIC);
    mqtt_subscribe(ILUMINATION_TOPIC);
    mqtt_subscribe(PROXIMITY_TOPIC);
    mqtt_subscribe(EMERGENCY_TOPIC);
}
```

Envío de Mensajes (enviarMensajePorTopic(const char topic, String outgoingMessage)): Esta función publica mensajes en un tópico MQTT específico. Toma como parámetros el tópico y el mensaje a enviar.

cpp

 Copiar código

```
void enviarMensajePorTopic(const char* topic, String outgoingMessage)
{
    mqtt_publish(topic, outgoingMessage.c_str()); // Publica el mensaje en el tópico MQTT
}
```


Recepción de Mensajes (alRecibirMensajePorTopic(char topic, String msg)): Esta función gestiona los mensajes recibidos en los tópicos a los que se ha suscrito. Los mensajes, excepto aquellos recibidos en HELLO_TOPIC y INCIDENCE_TOPIC, se almacenan en el buffer circular. Si el buffer está al 80% de su capacidad, los mensajes se listan, eliminan y se envían a través de INCIDENCE_TOPIC.

```
cpp Copiar código

void alRecibirMensajePorTopic(char* topic, String msg) {
    JsonDocument doc, vuelta, t;
    DeserializationError err = deserializeJson(doc, msg);
    String msg1;
    int nuevo_sensor;

    if(strcmp(topic, HELLO_TOPIC) != 0 && strcmp(topic, INCIDENCE_TOPIC) != 0)
    {
        put_item(msg, &Buffer); // Inserta el mensaje en el buffer
    }

    if ((float)Buffer.contador / Buffer.capacidad >= 0.8) // Comprueba si el buffer está al 80%
    {
        String result;
        listanddelete(&Buffer, result); // Lista y elimina elementos del buffer
        enviarMensajePorTopic(INCIDENCE_TOPIC, result); // Publica el resultado en MQTT
    }
}
```

Archivo set_up:

El archivo set_up se encarga de la configuración inicial de los pines y la creación de tareas para el funcionamiento del dispositivo. Esta configuración incluye la definición de los modos de los pines y el inicio de tareas FreeRTOS que gestionan sensores y botones. Además, envía un mensaje inicial a través de un tópico MQTT para indicar que el dispositivo está activo.

```
cpp Copiar código

void on_setup()
{
    pinMode(LED_BUILTIN, OUTPUT);
    pinMode(PIN_BUTTON, INPUT);
    pinMode(US_SENSOR1, OUTPUT);
    pinMode(US_SENSOR2, INPUT);

    String hello_msg = String("Hola Mundo! Desde dispositivo ") + deviceID;
    enviarMensajePorTopic(HELLO_TOPIC, hello_msg);

    xTaskCreate(illumination, "illumination", 2048, NULL, 1, &illumination_Task);
    xTaskCreate(readbutton, "readbutton", 2048, NULL, 1, &readbutton_Task);
    xTaskCreate(us, "us", 2048, NULL, 1, NULL);
}
```

RoboDK

En el contexto del desarrollo de aplicaciones en RoboDK, es fundamental comprender cómo diseñar y ejecutar programas que controlan el comportamiento de los robots de manera secuencial. En esta sección del manual, nos centraremos en la creación de programas secuenciales que guían las acciones de los robots a través de una serie de instrucciones predefinidas.

Es importante destacar que, a pesar de que los programas se ejecutan dentro de la simulación de RoboDK, todo el control y la lógica de ejecución residen en un entorno externo. Esto significa que podemos diseñar algoritmos complejos y sistemas de control avanzados fuera de la simulación y luego integrarlos con facilidad en nuestros programas de RoboDK.

A lo largo de esta sección, desglosaremos cada aspecto clave de la programación secuencial en RoboDK, desde la estructura básica de un programa hasta técnicas avanzadas para la gestión de flujo y la toma de decisiones. Además, proporcionaremos ejemplos prácticos y casos de uso para ilustrar cómo estos conceptos se aplican en situaciones reales.

MqttListener.py

Importaciones de librerías: Se importan las librerías necesarias para el funcionamiento del código, incluyendo la API de RoboDK, la librería MQTT, un módulo propio llamado RobotController para manejar las órdenes recibidas, y la librería json para trabajar con datos en formato JSON.

```
python Copiar código  
  
from robodk import robolink    # RoboDK API  
from robodk import robomath    # Robot toolbox  
import paho.mqtt.client as mqtt  
import RobotController as rc  
import json
```

Definición de topics MQTT: Se definen los nombres de los topics MQTT a los que se suscribirá el cliente MQTT para recibir mensajes relacionados con diferentes acciones y estados del sistema.

```
python Copiar código

DESPALETIZADO_COMMANDS_TOPIC = "giirob/pr2/B1/actions/despaletizado/commands"
ASIGNACION_COMMANDS_TOPIC = "giirob/pr2/B1/actions/asignacion/commands"
SENSOR1_COMMANDS_TOPIC = "giirob/pr2/B1/control/sensor1/commands"
CONVEYOR1_COMMANDS_TOPIC = "giirob/pr2/B1/actions/conveyor1/commands"
AGV_COMMANDS_TOPIC = "giirob/pr2/B1/actions/agv/commands"
INFOPEDIDO_TOPIC = "giirob/pr2/B1/infopedido"
NEWCONTAINER_TOPIC = "giirob/pr2/B1/newContainer"
EMERGENCY_TOPIC = "giirob/pr2/b1/control/emergency"
```

Configuración de conexión MQTT: Se establecen los parámetros de conexión al servidor MQTT, incluyendo la dirección, el puerto, el nombre de usuario y la contraseña, así como los tópicos base y específicos para enviar comandos y el estado del sistema.

```
python Copiar código

broker= "broker.hivemq.com"#"mqtt.dsic.upv.es"
port=1883
user="giirob"
passwd="UPV2024"
base_topic="giirob/pr2/B1"
station_commands_topic=base_topic+"/commands"
station_status_topic=base_topic+"/status"
```

Función de callback para manejar mensajes recibidos: Se define una función de callback que se ejecutará cada vez que se reciba un mensaje MQTT, extrayendo el payload, el tópico y el nivel de calidad de servicio (QoS) del mensaje recibido, y luego llamando a la función `handle_message` del módulo `RobotController`.

```
python Copiar código

def on_message(mqttc, obj, msg):
    payload = msg.payload.decode('utf-8')
    topic = msg.topic
    qos = msg.qos
    rc.handle_message(mqttc, topic, payload)
```

Configuración del cliente MQTT: Se crea una instancia del cliente MQTT, se configura para utilizar la versión 2 de la API de callback, se establece la función de callback para manejar los mensajes recibidos, se configuran las credenciales de conexión, se realiza la conexión al servidor MQTT y se suscribe el cliente a los diferentes tópicos definidos previamente.

```
python Copiar código

mqttc = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
mqttc.on_message = on_message

mqttc.username_pw_set(username=user, password=passwd)
mqttc.connect(broker, port, 60)
mqttc.subscribe(station_commands_topic, 0)
mqttc.subscribe(DESPALETIZADO_COMMANDS_TOPIC, 0)
mqttc.subscribe(ASIGNACION_COMMANDS_TOPIC, 0)
mqttc.subscribe(SENSOR1_COMMANDS_TOPIC, 0)
mqttc.subscribe(CONVEYOR1_COMMANDS_TOPIC, 0)
mqttc.subscribe(AGV_COMMANDS_TOPIC, 0)
mqttc.subscribe(INFOPEDIDO_TOPIC, 0)
mqttc.subscribe(NEWCONTAINER_TOPIC, 0)
mqttc.subscribe(EMERGENCY_TOPIC, 0)
```

Publicación del estado inicial: Se define el estado inicial del sistema en formato JSON y se publica en el tópico de estado del sistema.

```
python Copiar código

state_json = '{"STATE": "ready"}'
mqttc.publish(station_status_topic, state_json)
```

Bucle principal de escucha de mensajes MQTT: Se inicia el bucle principal del cliente MQTT, que se ejecutará indefinidamente y escuchará continuamente los mensajes entrantes, ejecutando la función de callback correspondiente cada vez que se reciba un mensaje.

```
python Copiar código

mqttc.loop_forever()
```

RobotController.py

El archivo "RobotController" desempeña un papel crucial en la gestión de las acciones del robot dentro de la simulación en RoboDK en respuesta a los mensajes recibidos a través del protocolo MQTT. Este archivo contiene una función principal llamada `handle_message` que se activa cada vez que se recibe un mensaje MQTT, determinando las acciones a tomar según el tópico del mensaje. Desde mover el robot hasta detener emergencias, cada acción se desencadena en función del tópico específico, lo que permite un control preciso y eficiente del comportamiento del robot en la simulación.

Importaciones de librerías: Se importan las librerías necesarias para el funcionamiento del código, incluyendo la API de RoboDK, la librería MQTT, un módulo propio llamado

`functions` para manejar las acciones del robot, y la librería `json` para trabajar con datos en formato JSON.

```
python Copiar código

from robodk import robolink # RoboDK API
from robodk import robomath # Robot toolbox
import json
import functions as fun
import paho.mqtt.client as mqtt

from robolink import * # API to communicate with RoboDK
from robodk import * # basic matrix operations
```

Definiciones de topics MQTT: Se definen los nombres de los topics MQTT que serán manejados por el controlador de robot, incluyendo tópicos para diferentes acciones y estados del sistema.

```
python Copiar código

DESPALETIZADO_COMMANDS_TOPIC = "giirob/pr2/B1/actions/despaletizado/commands"
ASIGNACION_COMMANDS_TOPIC = "giirob/pr2/B1/actions/asignacion/commands"
SENSOR1_COMMANDS_TOPIC = "giirob/pr2/B1/control/sensor1/commands"
CONVEYOR1_COMMANDS_TOPIC = "giirob/pr2/B1/actions/conveyor1/commands"
AGV_COMMANDS_TOPIC = "giirob/pr2/B1/actions/agv/commands"
INFOPEDIDO_TOPIC = "giirob/pr2/B1/infopedido"
NEWCONTAINER_TOPIC = "giirob/pr2/B1/newContainer"
EMERGENCY_TOPIC = "giirob/pr2/b1/control/emergency"
```

Función `handle_message`: Se define una función llamada `handle_message` que será llamada cada vez que se reciba un mensaje MQTT. Esta función recibe tres parámetros: `mqttc`, el cliente MQTT; `topic`, el tóptico del mensaje recibido; y `payload`, el contenido del mensaje.

```
python Copiar código

def handle_message(mqttc, topic, payload):
```

Bloques `if` para procesar mensajes según el tóptico: Se utilizan bloques `if` para verificar si el `topic` del mensaje coincide con uno de los tópicos definidos. Dependiendo del tóptico, se llama a una función específica del módulo `functions` para ejecutar la acción correspondiente en el simulador de RoboDK.

```
python Copiar código

if topic == DESPALETIZADO_COMMANDS_TOPIC:
    fun.move_robot_getBox(payload)
```

Acciones para el t3pico de emergencia: En caso de recibir un mensaje en el t3pico de emergencia (`EMERGENCY_TOPIC`), se detienen varios scripts en ejecuci3n mediante el m3todo `Stop()` y se muestra un mensaje de emergencia en la interfaz de RoboDK.

```
python Copiar c3digo  
  
if topic == EMERGENCY_TOPIC:  
    RDK.ShowMessage("EMERGENCY STOP")  
    scripts_to_stop = ['MqttListener', 'Obtener_info', 'orderControl', 'RunConveyor1']  
    for script_name in scripts_to_stop:  
        py_script = RDK.Item(script_name)  
        py_script.Stop()
```

Functions.py

El archivo "functions" contiene una serie de funciones dise1adas para ejecutar acciones espec3ficas en la simulaci3n de RoboDK en respuesta a mensajes recibidos a trav3s del protocolo MQTT. Estas funciones est3n destinadas a controlar el comportamiento del robot y otros elementos del entorno virtual, como cintas transportadoras y sensores. Cada funci3n se encarga de una tarea particular, como mover el robot a una posici3n espec3fica, activar un sensor o detener una cinta transportadora. Este archivo desempe1a un papel esencial en la coordinaci3n de las acciones del sistema dentro de la simulaci3n, garantizando un control preciso y eficiente de los elementos rob3ticos y de automatizaci3n.

Funciones para ejecutar programas en la simulaci3n:

- **move_robot_setBox(data):** Mueve el robot a una posici3n espec3fica para colocar una caja, utilizando datos de posici3n proporcionados.
- **move_robot_getBox(payload):** Ejecuta un programa para que el robot recoja una caja en la simulaci3n, utilizando un payload JSON como entrada.
- **move_conveyor1(payload):** Inicia o detiene la cinta transportadora en la simulaci3n seg3n el programa especificado en el payload JSON.
- **start_Sensor1():** Activa un sensor en la simulaci3n.
- **move_agv(program):** Ejecuta programas relacionados con el AGV (veh3culo de guiado autom3tico) en la simulaci3n, como iniciar, colocar o finalizar una tarea.

Funciones para enviar mensajes MQTT:

- **publishDespaletizado():** Publica un mensaje indicando que el proceso de despaletizado est3 listo en el t3pico correspondiente.
- **publishSensor1():** Publica mensajes para indicar el estado del sensor en la simulaci3n en los t3picos correspondientes.

- **publishAsignacion():** Publica un mensaje indicando que el proceso de asignación está listo en el tópico correspondiente.
- **webInfo(payload):** Guarda información recibida en formato JSON en un archivo de texto en la ubicación especificada.

Otras funciones:

- **parse_coordinates(coordinate_string):** Convierte una cadena de coordenadas en formato de cadena a una tupla de números.
- **deserializeJSONaction(payload):** Deserializa el payload JSON para obtener el nombre de un programa a ejecutar.
- **stopConveyor():** Detiene la cinta transportadora en la simulación.

Los siguientes cuatro y últimos programas en esta sección desempeñan funciones fundamentales en el entorno de simulación. Cada uno está diseñado para una tarea específica: la generación de nuevos objetos, la despaletización de cajas, la colocación de cajas y el restablecimiento de valores. Son modularizados y representan un enfoque eficaz para gestionar diversos aspectos de la simulación, desde la creación de elementos hasta el control de procesos, lo que permite una organización clara y una fácil expansión según las necesidades del proyecto.

Colocar_”.py

Este programa se encarga de la colocación precisa de cajas en posiciones específicas dentro del entorno de simulación en RoboDK. Comienza moviendo el robot a una posición inicial predeterminada, luego se desplaza a una posición de preparación para recoger una caja. Una vez que el robot está en posición, utiliza un gripper para agarrar la caja y la transporta a la ubicación de destino deseada. Después de soltar la caja en su posición final, el robot regresa a su posición inicial. Este programa es esencial para automatizar el proceso de colocación de cajas en aplicaciones de manufactura y logística simuladas, garantizando una colocación precisa y eficiente de los objetos.

Definición de Posiciones y Objetos: El script comienza definiendo diversas posiciones y objetos necesarios para el proceso. Esto incluye la posición inicial del robot (`pose_home`), la posición de recogida de la caja (`pose_pick_XL`), la posición de preparación para recoger la caja (`pose_prepick_XL`), la posición de destino para la caja (`pose_place`), el gripper (`tool`), y los marcos de referencia (`system_ref` y `system_ref_place`).

```
python
Copiar código

robot = RDKit.Item('UR20', ITEM_TYPE_ROBOT)
pose_place = RDKit.Item('Pose_place', ITEM_TYPE_TARGET)
pose_pick_XL = RDKit.Item('Pick_XL', ITEM_TYPE_TARGET)
pose_prepick_XL = RDKit.Item('PrePick_XL', ITEM_TYPE_TARGET)
pose_home = RDKit.Item('HOME_UR', ITEM_TYPE_TARGET)
tool = RDKit.Item('Vacuum Gripper cobot', ITEM_TYPE_TOOL)
system_ref = RDKit.Item('UR20 Base', ITEM_TYPE_FRAME)
system_ref_place = RDKit.Item('containerFrame', ITEM_TYPE_FRAME)
place = pose_place.Pose()
```

Activación y Desactivación del Control de TCP: Las funciones `TCP_On()` y `TCP_Off()` se utilizan para activar y desactivar el control del punto de herramienta (TCP, Tool Center Point). Esto asegura que el gripper se pueda conectar y desconectar correctamente durante la ejecución del programa.

```
python Copiar código  
  
def TCP_On(toolitem):  
    toolitem.AttachClosest()  
    toolitem.RDK().RunMessage('Set air valve on')  
    toolitem.RDK().RunProgram('TCP_On()')  
  
def TCP_Off(toolitem, itemleave=0):  
    toolitem.DetachAll(itemleave)  
    toolitem.RDK().RunMessage('Set air valve off')  
    toolitem.RDK().RunProgram('TCP_Off()')
```

Control del Gripper: Antes de recoger y colocar la caja, se activa el control del TCP utilizando la función `TCP_On()`. Una vez que la caja se ha colocado en su destino, el control del TCP se desactiva utilizando la función `TCP_Off()`.

```
python Copiar código  
  
robot.setPoseFrame(system_ref)  
  
robot.MoveJ(pose_home)  
robot.MoveJ(pose_prepick_XL)  
robot.MoveL(pose_pick_XL)  
TCP_On(tool)  
robot.MoveL(pose_prepick_XL)  
robot.MoveJ(pose_home)  
robot.MoveJ(place)  
TCP_Off(tool, system_ref_place)  
robot.MoveJ(pose_home)
```

Publicación de Mensaje de Asignación: Finalmente, se publica un mensaje indicando que el proceso de asignación está listo utilizando la función `publishAsignacion()` del módulo `functions`. Esto puede ser útil para coordinar actividades adicionales dentro de la simulación o con sistemas externos.

```
python Copiar código  
  
f.publishAsignacion()
```


Despaletizar_”.py

Este programa desempeña la tarea de despaletizar cajas en la simulación de RoboDK. Utiliza movimientos precisos del robot para extraer cajas de una paleta y colocarlas en una ubicación deseada. Controla la activación del TCP para el gripper y puede emitir un mensaje al completar el proceso. Es esencial para la automatización de tareas de manipulación de materiales en la simulación.

Definición de Posiciones y Objetos: Se definen varias posiciones y objetos necesarios para el proceso de despaletizado, incluyendo la posición de recogida de cajas, la posición de trabajo, la herramienta (gripper) y marcos de referencia.

Cálculo de Posiciones: Se calculan las posiciones de recogida y colocación de las cajas ajustando la posición de recogida base.

```
python Copiar código

robot = RDK.Item('ABB IRB 6640-130/3.2', ITEM_TYPE_ROBOT)
tool = RDK.Item('OnRobot VGC10 Vacuum Gripper', ITEM_TYPE_TOOL)
frame_place = RDK.Item('Move_conveyor1', ITEM_TYPE_FRAME)

desplazamientoX = RDK.getParam('PalletXL_Width', str_type=True)
desplazamientoY = RDK.getParam('PalletXL_Length', str_type=True)
desplazamientoZ = RDK.getParam('PalletXL_Heigth', str_type=True)
N_BXL = RDK.getParam('Cont_BXL', str_type=True)

pick_XL = RDK.Item('pose_pick_XL', ITEM_TYPE_TARGET)
paso_XL = RDK.Item('pose_paso_XL', ITEM_TYPE_TARGET)
place_XL_L = RDK.Item('pose_place_XL_L', ITEM_TYPE_TARGET)
pre_place_XL_L = RDK.Item('pose_preplace_XL_L', ITEM_TYPE_TARGET)
pose_pick_XL = pick_XL.Pose()

pose_pick_XL = pose_pick_XL * transl(int(desplazamientoX), int(desplazamientoY), int(despl
pose_prepick_XL = pose_pick_XL * transl(0, 0, -50)
pose_paso_XL = paso_XL.Pose()
pose_place_XL_L = place_XL_L.Pose()
pose_preplace_XL_L = pre_place_XL_L.Pose()
```

Activación y Desactivación del Control del TCP: Se activa el control del TCP para la herramienta antes de la recogida y se desactiva después de la colocación de la caja.

```
python Copiar código

def TCP_On(toolitem):
    toolitem.AttachClosest()
    toolitem.RDK().RunMessage('Set air valve on')
    toolitem.RDK().RunProgram('TCP_On()');

def TCP_Off(toolitem, itemleave=0):
    toolitem.DetachAll(frame_place)
    toolitem.RDK().RunMessage('Set air valve off')
    toolitem.RDK().RunProgram('TCP_Off()');
```

Movimiento del Robot: El robot se mueve secuencialmente a través de una serie de posiciones predeterminadas: desde la posición de recogida hasta la posición de colocación, pasando por una posición de trabajo intermedia.

```
python Copiar código

robot.MoveJ(pose_preplace_XL_L)
robot.MoveJ(pose_paso_XL)
robot.MoveJ(pose_prepick_XL)
robot.MoveL(pose_pick_XL)
TCP_On(tool)
robot.MoveL(pose_prepick_XL)
robot.MoveJ(pose_paso_XL)
robot.MoveJ(pose_preplace_XL_L)
robot.MoveJ(pose_place_XL_L)
TCP_Off(tool, frame_place)
robot.MoveJ(pose_preplace_XL_L)
```

Ajuste de Parámetros: Se ajustan los parámetros relacionados con las dimensiones de la paleta después de cada ciclo de despaletizado.

```
python Copiar código

RDK.setParam('PalletXL_Width', int(desplazamientoX) - 250)
if int(desplazamientoX) == -500:
    RDK.setParam('PalletXL_Width', 0)
    RDK.setParam('PalletXL_Length', int(desplazamientoY) + 300)
    if int(desplazamientoY) == 900:
        RDK.setParam('PalletXL_Length', 0)
        RDK.setParam('PalletXL_Heigth', int(desplazamientoZ) + 210)
        if int(desplazamientoZ) == 420:
            RDK.setParam('PalletXL_Heigth', 0)
RDK.setParam('Cont_BXL', N_BXL - 1)
```

Publicación de Mensajes: Se publica un mensaje para indicar que el proceso de despaletizado ha sido completado.

Control de Flujo: Si se ha completado el despaletizado de todas las cajas de la paleta, se ejecuta un programa para generar un nuevo pallet.

```
python Copiar código

pause(2)
f.publishDespaletizado()

N_BXL = RDK.getParam('Cont_BXL')

if N_BXL == 0:
    py_script = RDK.Item('Nuevo_Pallet_XL')
    py_script.RunProgram()
```

Reset_”.py

Este programa desempeña una función crucial en el entorno de simulación de RoboDK al restablecer valores y condiciones específicas después de completar ciertos procesos o ciclos. Es esencial para mantener la coherencia y eficiencia del sistema robótico simulado, garantizando que esté listo para iniciar nuevas operaciones o ciclos de producción. El programa de reseteo asegura que todas las variables y parámetros relevantes vuelvan a su estado inicial, preparando el entorno para futuras tareas de manera ordenada y controlada.

El programa de reseteo desempeña una función crítica en la simulación de RoboDK al restaurar valores y condiciones específicas a su estado inicial. Es esencial para mantener la consistencia y preparar el entorno para futuras tareas. Este programa restablece variables y parámetros relevantes a valores predefinidos, asegurando que el sistema robótico simulado esté listo para iniciar nuevos ciclos de producción de manera ordenada y controlada.

```
python Copiar código

RDK.setParam('PalletXL_Width', 0)
RDK.setParam('PalletXL_Length', 0)
RDK.setParam('PalletXL_Heigth', 0)
RDK.setParam('Cont_BXL', 36)
```

Nuevo_Pallet_”.py

El programa "Nuevo Pallet" cumple una función crucial en la simulación de RoboDK al generar nuevos pallets o plataformas de trabajo. Estos pallets son fundamentales para la organización y manipulación de objetos dentro del entorno simulado, permitiendo una gestión eficiente de la producción o logística virtual. Este programa se encarga de crear un

nuevo pallet con las dimensiones y características adecuadas, listo para recibir objetos o productos y facilitar su manipulación por parte de los robots simulados.

Inicialización y Configuración: Esta sección establece la conexión con RoboDK y realiza la configuración inicial necesaria para el programa.

```
python Copiar código

from robodk import *      # RoboDK API
from robolink import *    # Robot toolbox
RDK = Robolink()

RDK.setParam('Cont_BS', 192)
py_script = RDK.Item('Reset_param_S')
py_script.RunProgram()
```

Eliminación de Pallet Existente: Esta parte del código elimina cualquier pallet existente en la estación antes de crear uno nuevo.

```
python Copiar código

padre = RDK.Item('STATION', ITEM_TYPE_FRAME)
child_objects = padre.Children()
for obj in child_objects:
    if obj.Name() == 'PalletS':
        obj.Delete()
```

Creación de Nuevo Pallet: Aquí se realiza la copia y pegado de un pallet existente para crear uno nuevo en la estación.

```
python Copiar código

pallets = RDK.Item('PalletS', ITEM_TYPE_OBJECT)
pallets.Copy()
newpallets = RDK.Paste()
newpallets.setParent(padre)
newpallets.setVisible(1)
```

RoboDK es una herramienta poderosa y versátil que ofrece una solución integral para la simulación y programación de robots. Su interfaz intuitiva y su amplia gama de funciones facilitan la creación y la optimización de procesos robóticos en entornos industriales y de investigación. Desde la simulación 3D hasta la generación de código robot, RoboDK permite a los usuarios diseñar, visualizar y validar aplicaciones robóticas de manera eficiente y precisa. Además, su compatibilidad con una amplia variedad de robots y controladores lo convierte en una opción ideal para proyectos de cualquier escala y complejidad. En resumen, RoboDK es una herramienta indispensable para aquellos que buscan mejorar la productividad y la eficiencia en el desarrollo de aplicaciones robóticas.

Conexiones y Comunicaciones

Configuración y conexión de la API REST

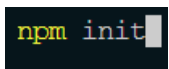
Para conectar un servidor con una web, necesitamos de código de conexión tanto en el backend como en el frontend.

BACKEND

Para nuestro backend, hemos configurado una API REST muy sencilla. En primer lugar, hay que entender lo que es; la API REST (Representational State Transfer) es un conjunto de reglas y convenciones para el intercambio de datos entre sistemas a través de HTTP. Utiliza los métodos de solicitud HTTP: GET, POST, PUT y DELETE para realizar operaciones en recursos. Se basa en la arquitectura cliente-servidor y es altamente escalable y flexible. Las respuestas suelen estar en formato JSON o XML. Se usa mucho en aplicaciones web y móviles para permitir la comunicación entre cliente y servidor de manera eficiente y sin estado.

Así, con una API REST podremos conectar y mandar pedidos desde nuestra web al servidor.

Para ello, primero necesitamos inicializar el npm para gestionar y administrar paquetes de software en entornos de desarrollo de JavaScript, y lo hacemos de la siguiente manera en la terminal:



Este comando generará un archivo package.json en el directorio raíz de tu proyecto. Este archivo package.json es fundamental para el proyecto, ya que contiene información sobre las dependencias, scripts y configuraciones del proyecto como podemos ver en la siguiente imagen:

Una vez inicializadas todas las dependencias y librerías creamos dos archivos: **index.js** y **database.js**.

El index.js va a ser el “main” de nuestro servidor, donde además de tener las inicializaciones pertinentes, tenemos dos solicitudes HTTP.

Un GET para obtener la información del pedido y un POST que recibe la información del pedido para introducirla en la BBDD.

```
{
  "name": "api_sql",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "dev": "nodemon src/index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "morgan": "^1.10.0",
    "nodemon": "3.1.0"
  },
  "dependencies": {
    "cors": "^2.8.5",
    "express": "4.19.2",
    "pg": "^8.11.5"
  }
}
```

```

// Importar las bibliotecas necesarias
const express = require('express');
const cors = require('cors');
const morgan = require('morgan');

// Crear una instancia de la aplicación Express
const app = express();

// Configuración inicial
app.set('port', 4000); // Establece el puerto en 4000
app.listen(app.get('port')); // Comienza a escuchar en el puerto especificado
console.log("Escuchando en el puerto " + app.get('port')); // Imprime en consola el puerto

// Middleware
app.use(cors({
  origin: ["http://127.0.0.1:5501", "http://127.0.0.1:5500"]
})); // Habilita CORS para los orígenes especificados
app.use(morgan('dev')); // Utiliza morgan para el registro de solicitudes HTTP en la consola
app.use(express.json()); // Permite que la aplicación maneje JSON

// Rutas
app.get("/pedido", async (req, res) => {
  const connection = database; // Obtiene la conexión a la base de datos
  const result = await connection.query("SELECT * FROM pedido"); // Ejecuta la consulta
  res.json(result); // Envía el resultado como respuesta en formato JSON
});

```

```

app.post("/pedido", async (req, res) => {
  if (req.body && req.body.length > 0) {
    let totalPrice = 0;
    let cajas_s = 0;
    let cajas_m = 0;
    let cajas_l = 0;
    let cajas_xl = 0;

    // Calcula el precio total y la cantidad de cajas de cada tipo
    req.body.forEach(producto => {
      for (let i = 0; i < producto.length; i++) {
        totalPrice += parseInt(producto[i].price);
        if (producto[i].title == "BOX S") {
          cajas_s = parseInt(producto[i].quantity);
        } else if (producto[i].title == "BOX M") {
          cajas_m = parseInt(producto[i].quantity);
        } else if (producto[i].title == "BOX L") {
          cajas_l = parseInt(producto[i].quantity);
        } else if (producto[i].title == "BOX XL") {
          cajas_xl = parseInt(producto[i].quantity);
        }
      }
    });
  }
});

```

```

// Prepara la consulta para insertar un nuevo pedido
const consultaPedido = {
  text: "INSERT INTO pedido (cajas_s, cajas_m, cajas_l, cajas_xl, precio_total) '
  values: [cajas_s, cajas_m, cajas_l, cajas_xl, TotalPrice]
};

try {
  const result = await database.query(consultaPedido); // Ejecuta la consulta de
  function devolverIdPedido() {
    return result.rows[0].id_pedido; // Devuelve el ID del pedido insertado
  }
} catch (err) {
  console.log(err.stack); // Imprime cualquier error en la consola
}

return res.sendStatus(200); // Envía un estado 200 (OK) como respuesta
}

res.sendStatus(400); // Envía un estado 400 (Bad Request) si el cuerpo de la solicitud
window.location.href = "fallo_conexion.html"; // Redirige a una página de fallo de con
});

```

Aparte, en el archivo **database.js** introducimos los atributos necesarios para conectarnos a nuestra base de datos PostgreSQL.

```

const { Pool } = require('pg');

const database = new Pool({
  user: "postgres",
  host: "localhost",
  database: "postgres",
  password: "Gerfederer44",
});

module.exports = database;

```

*NOTA -> Hacemos un export default de la función “database” para luego llamarla en el **index.js** donde sea necesario.

FRONTEND

De la parte del FrontEnd, creamos un archivo JS llamado **pedido_a_BBDD.js** en el cual se hace una solicitud POST al servidor con la información del pedido que está almacenada en una variable llamada “products” del local storage.

Este archivo, crea una conexión con el código del backend antes comentado, cabe recalcar, que para que funcione la conexión necesitamos tener la API REST corriendo en todo momento para que pueda escuchar solicitudes.

*NOTA -> Si queremos desplegar la web en la nube, será necesario que el backend, es decir, la API la tengamos subida a un servidor web para poder hacer solicitudes de usuarios distintos al localhost

```
document.getElementById("confirmar-compra").addEventListener("click", async () => {
  const carrito = JSON.parse(localStorage.getItem("products"));
  if (carrito && carrito.length > 0) {
    try {
      const res = await fetch("https://nn-almacenes-inteligentes.netlify.app/carrito", {
        method: "POST",
        headers: {
          "Content-Type": "application/json"
        },
        body: JSON.stringify(carrito)
      });
      if (res.ok) {
        console.log("Compra realizada");
      } else {
        window.location.href = "fallo_conexion.html";
      }
    } catch (error) {
      console.error("Error al realizar la compra:", error);
      window.location.href = "fallo_conexion.html";
    }
  }
});
```

Configuración y conexión MQTT

Hemos optado por el uso del protocolo MQTT para la conexión del conocido Internet de las Cosas, es decir, toda la parte de conexiones entre el algoritmo C++ y los ESP32, al igual que los ESP32 y robodk y por último la web con el algoritmo.

Web

Para la parte de la web, necesitamos una conexión MQTT para enviar la información necesaria del pedido a un topic donde el algoritmo en C++ leerá los datos y calculará las posiciones de las cajas en el contenedor.

Para ello, necesitamos un archivo en el frontend llamado **mqtt.js** en el que, con el uso de la librería para MQTT en la web comentada en el apartado de dependencias y librerías necesarias, nos conectamos a un Broker MQTT (podéis gastar cualquier broker público o de pago existente), en nuestro caso, hemos utilizado el broker público: "wss://broker.emqx.io:8084/mqtt", y luego, nos conectamos al broker, nos suscribimos al tópico pertinente (giirob/pr2/B1/infopedido) y publicamos la información del pedido en formato JSON en ese tópico.

Aquí os dejamos una imagen del archivo **mqtt.js**:

```
const url = "wss://broker.emqx.io:8084/mqtt";
var client = mqtt.connect(url);

const carritoElement = document.getElementById("confirmar-compra");

async function PublicarPedido() {
  client.on('connect', function () {
    carritoElement.addEventListener('click', () => {
      const productos = JSON.parse(localStorage.getItem("products"));
      for (let i = 0; i < productos.length; i++) {
        delete productos[i].img;
        delete productos[i].price;
        delete productos[i].title;
        delete productos[i].size;
      }
      if (productos && productos.length > 0) {
        client.subscribe('giirob/pr2/B1/infopedido', function (err) {
          if (!err) {
            client.publish('giirob/pr2/B1/infopedido', JSON.stringify(productos));
            if (err) {
              window.location.href = "fallo_conexion.html";
            }
          }
        });
      } else {
        window.location.href = "fallo_conexion.html";
      }
    });
  } else {
    console.log("No hay productos en el carrito");
  }
});
```

Algoritmo

A continuación, se presenta el fichero mqtt.cpp en C++ para conectarse a un servidor MQTT utilizando la biblioteca Paho MQTT C++. Este ejemplo incluye la conexión al servidor, la publicación de un mensaje y la suscripción a un tópico.

Primero, asegúrate de tener instalada la biblioteca Paho MQTT C++ y sus dependencias. Puedes encontrar la documentación y las instrucciones de instalación en Paho MQTT C++.

El código comienza incluyendo varias bibliotecas necesarias para trabajar con MQTT en C++. Define constantes para la dirección del servidor MQTT, el ID del cliente, el tema de

suscripción y parámetros de calidad de servicio (QoS) y reintentos. Estas configuraciones iniciales establecen la base para la conexión y la suscripción al servidor MQTT.

Inclusión de Bibliotecas

```
cpp Copiar código

#include <iostream>
#include <cstdlib>
#include <string>
#include <cstring>
#include <cctype>
#include <thread>
#include <chrono>
#include "../paho.mqtt.c++/mqtt/async_client.h"
```

Se incluyen las bibliotecas estándar de C++ y la biblioteca Paho MQTT C++.

Constantes

```
cpp Copiar código

const std::string SERVER_ADDRESS("mqtt://localhost:1883");
const std::string CLIENT_ID("paho_cpp_async_subscribe");
const std::string TOPIC("hello");

const int QOS = 1;
const int N_RETRY_ATTEMPTS = 5;
```

Define la dirección del servidor MQTT, el ID del cliente, el tópico al que se suscribe, la calidad de servicio (QoS) y el número de intentos de reconexión.

Clases y funciones adaptadas y utilizadas de la librería paho.mqtt.cpp

Clase `action_listener`: Esta clase maneja la llegada de mensajes, la pérdida de conexión y la reconexión. Implementa varias funciones de callback para estos eventos.

Clase `callback`: Esta clase maneja la llegada de mensajes, la pérdida de conexión y la reconexión. Implementa varias funciones de callback para estos eventos.

Función `reconnect()`: Intenta reconectar al servidor MQTT si la conexión se pierde.

Función `on_failure()`: Maneja fallos de conexión e intenta reconectar.

Función `on_success()`: Maneja el éxito de las acciones, aunque no hace nada en este caso.

Función `connected()`: Se llama cuando la conexión es exitosa y suscribe al cliente al tópico.

Función `connection_lost()`: Maneja la pérdida de conexión y trata de reconectar.


Función `message_arrived()`: Se llama cuando un mensaje llega al tópico suscrito.

Main de la conexión:

Creación del cliente MQTT y Configuración de las opciones de conexión:


Se crea un objeto `mqtt::async_client` utilizando la dirección del servidor y el ID del cliente. También se configura un objeto `mqtt::connect_options` para especificar que no se debe limpiar la sesión al desconectarse, lo que permite que el servidor recuerde el cliente y sus suscripciones. Esta sección prepara el cliente y las opciones de conexión necesarias para establecer una conexión persistente con el servidor MQTT.

cpp

 Copiar código

```
mqtt::async_client cli(SERVER_ADDRESS, CLIENT_ID);
```

cpp


 Copiar código

```
mqtt::connect_options connOpts;  
connOpts.set_clean_session(false);
```

Configuración del Callback:

El programa intenta conectar al servidor MQTT utilizando las opciones de conexión configuradas y el callback. Si la conexión falla, se captura la excepción y se muestra un mensaje de error. Este bloque de código es crucial ya que establece la conexión inicial al servidor, permitiendo que el cliente empiece a recibir mensajes y eventos.

cpp

 Copiar código

```
callback cb(cli, connOpts);  
cli.set_callback(cb);
```

Conexión al servidor MQTT:

El programa intenta conectar al servidor MQTT utilizando las opciones de conexión configuradas y el callback. Si la conexión falla, se captura la excepción y se muestra un mensaje de error. Este bloque de código es crucial ya que establece la conexión inicial al servidor, permitiendo que el cliente empiece a recibir mensajes y eventos.

cpp


 Copiar código

```
try {
    std::cout << "Connecting to the MQTT server..." << std::flush;
    cli.connect(connOpts, nullptr, cb);
}
catch (const mqtt::exception& exc) {
    std::cerr << "\nERROR: Unable to connect to MQTT server: '"
                << SERVER_ADDRESS << "'" << exc << std::endl;
    return 1;
}
```

Bucle principal para mantener la aplicación en ejecución:

Una vez conectado, el programa entra en un bucle que espera hasta que el usuario ingrese la letra 'q' para finalizar. Este bucle permite que el cliente siga recibiendo y procesando mensajes mientras el programa esté en ejecución. Es una manera sencilla de mantener el programa activo sin realizar otras tareas.

cpp


 Copiar código

```
while (std::tolower(std::cin.get()) != 'q')
    ;
```

Desconexión del servidor MQTT:

Finalmente, cuando el usuario decide salir, el programa intenta desconectarse del servidor MQTT. Si ocurre algún error durante la desconexión, se captura y se muestra una excepción. Este paso asegura que el cliente cierre la conexión de manera ordenada y libere los recursos asociados.

cpp

 Copiar código

```
try {
    std::cout << "\nDisconnecting from the MQTT server..." << std::flush;
    cli.disconnect()->wait();
    std::cout << "OK" << std::endl;
}
catch (const mqtt::exception& exc) {
    std::cerr << exc << std::endl;
    return 1;
}
```

ESP32S3 MASTER Y SLAVE

Inclusión de bibliotecas:

Aquí se incluyen las bibliotecas necesarias para el funcionamiento del dispositivo. Estas bibliotecas proporcionan funcionalidades para la conexión WiFi, comunicación MQTT, manipulación de JSON, control de pantallas LCD, entre otras.

```
cpp Copiar código

#include "Config.h"
#include "order.h"
#include <WiFi.h>
#ifdef SSL_ROOT_CA
#include <WiFiClientSecure.h>
#endif
#include <PubSubClient.h>
#include <ArduinoJson.h>
#include <LiquidCrystal_I2C.h>
```

Función de configuración (setup()):

La función `setup()` es el punto de entrada del programa Arduino. Aquí se realiza la configuración inicial del dispositivo. Esto incluye la inicialización de la comunicación serial (si está habilitada), la conexión WiFi, la conexión MQTT, la suscripción a los temas relevantes, y la ejecución de otros métodos de inicialización.

Serialización (opcional): Si la macro `LOGGER_ENABLED` está definida, se inicializa la comunicación serial con una velocidad específica (`BAUDS`). Esto es útil para la depuración y el registro de información durante la ejecución del programa. Se agrega un breve retraso (`delay(1000)`) y se imprime una línea en blanco en el puerto serie.


```
cpp Copiar código

void setup() {
#ifdef LOGGER_ENABLED
    // Inicializamos comunicaciones serial
    Serial.begin(BAUDS);
    delay(1000);
    Serial.println();
#endif
}
```

Conexión WiFi:

Se llama a la función `wifi_connect()` para establecer la conexión WiFi del dispositivo. Esta función debe estar definida en otra parte del código y se encarga de configurar y conectar el dispositivo a la red WiFi especificada.

cpp


 Copiar código

```
// Nos conectamos a la wifi  
wifi_connect();
```

Conexión MQTT:

Se llama a la función `mqtt_connect()` para conectarse al broker MQTT. Se pasa como parámetro el `deviceId`, que se utilizó para identificar de forma única al dispositivo. Esta función debe estar definida en otro lugar del código y se encarga de establecer la conexión con el servidor MQTT.

cpp


 Copiar código

```
// Nos conectamos al broker MQTT, indicando un 'client-id'  
mqtt_connect(deviceID);
```

Suscripción a tópicos MQTT:

Se llama a la función `suscribirseATopics()`, que aparentemente está definida en otro archivo (probablemente `g_comunicaciones.ino`). Esta función se encarga de suscribir al dispositivo a los tópicos MQTT relevantes para recibir mensajes.

cpp


 Copiar código

```
// TODO: completar esta función (g_comunicaciones.ino)  
suscribirseATopics();
```

Inicialización específica del dispositivo:

Se llama a la función `on_setup()`, que aparentemente está definida en otro archivo (probablemente `s_setup.ino`). Esta función se encarga de realizar cualquier inicialización específica del dispositivo, como configuración de pines, inicialización de variables, etc.

cpp

 Copiar código

```
// TODO: completar esta función (s_setup.ino)  
on_setup();
```

Ajuste del tamaño del buffer MQTT:

Se llama a la función `mqtt_resize_buffer()`, que probablemente ajusta el tamaño del buffer utilizado para recibir y enviar mensajes MQTT. Esto puede ser necesario para adaptarse a los requisitos específicos de la aplicación.

cpp

 Copiar código

```
mqtt_resize_buffer();
```


robodk

mqttlistener

Importación de bibliotecas:


Aquí se importan las bibliotecas necesarias para utilizar la API de RoboDK, que se utiliza para controlar un robot. Se importan dos módulos específicos: `robolink` para la comunicación con RoboDK y `robomath` para operaciones matemáticas relacionadas con robots. Se importan la biblioteca `paho.mqtt.client` para la comunicación MQTT y el módulo `RobotController` (presumiblemente definido en otro archivo) para manejar el control del robot.

python

 Copiar código

```
from robodk import robolink    # RoboDK API
from robodk import robomath    # Robot toolbox
```

python

 Copiar código

```
import paho.mqtt.client as mqtt
import RobotController as rc
```

Definición de tópicos MQTT:

Se definen los topics MQTT que se utilizarán para la comunicación entre el dispositivo y el broker MQTT. Estos tópicos representan diferentes comandos y eventos relacionados con las acciones del robot y el estado del sistema.

python

 Copiar código

```
DESPALETIZADO_COMMANDS_TOPIC = "giirob/pr2/B1/actions/despaletizado/commands"
ASIGNACION_COMMANDS_TOPIC = "giirob/pr2/B1/actions/asignacion/commands"
SENSOR1_COMMANDS_TOPIC = "giirob/pr2/B1/control/sensor1/commands"
CONVEYOR1_COMMANDS_TOPIC = "giirob/pr2/B1/actions/conveyor1/commands"
AGV_COMMANDS_TOPIC = "giirob/pr2/B1/actions/agv/commands"
INFOPEDIDO_TOPIC = "giirob/pr2/B1/infopedido"
NEWCONTAINER_TOPIC = "giirob/pr2/B1/newContainer"
EMERGENCY_TOPIC = "giirob/pr2/b1/control/emergency"
```

Configuración del cliente MQTT:

Se configuran los parámetros de conexión al broker MQTT, incluyendo la dirección IP o nombre de dominio (`broker`), el puerto (`port`), el nombre de usuario (`user`) y la contraseña (`passwd`). También se definen los tópicos de comandos y estado del dispositivo.

```
python                                                                    Copiar código

broker = "broker.hivemq.com"#"mqtt.dsic.upv.es"
port = 1883
user = "giirob"
passwd = "UPV2024"
base_topic = "giirob/pr2/B1"
station_commands_topic = base_topic + "/commands"
station_status_topic = base_topic + "/status"
```

Definición de la función de manejo de mensajes MQTT:

Se define la función `on_message()` que se llamará cada vez que se reciba un mensaje MQTT. Esta función decodifica el payload del mensaje y llama a la función `handle_message()` del módulo `RobotController` para manejar el mensaje recibido.

```
python                                                                    Copiar código

def on_message(mqttc, obj, msg):
    payload = msg.payload.decode('utf-8')
    topic = msg.topic
    qos = msg.qos
    rc.handle_message(mqttc, topic, payload)
```

Inicialización y conexión del cliente MQTT:

Se crea una instancia del cliente MQTT y se configura para utilizar la versión 2 de la API de devolución de llamada de MQTT. Se establecen el nombre de usuario y la contraseña para la conexión al broker MQTT. Luego, se conecta al broker y se suscribe a los tópicos de comandos relevantes.

```
python                                                                    Copiar código


mqttc = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
mqttc.on_message = on_message

mqttc.username_pw_set(username=user, password=passwd)
mqttc.connect(broker, port, 60)
mqttc.subscribe(station_commands_topic, 0)
mqttc.subscribe(DESPALETIZADO_COMMANDS_TOPIC, 0)
# Se subscribe a otros tópicos aquí...
```


Publicación del estado inicial del dispositivo:

Se define un objeto JSON que representa el estado inicial del dispositivo (en este caso, listo). Luego, se publica este estado en el tópico `station_status_topic`.

python

 Copiar código

```
state_json = '{"STATE": "ready"}'  
mqttc.publish(station_status_topic, state_json)
```

En conclusión, el manual de programación proporciona una guía completa y detallada para los usuarios sobre cómo utilizar y entender el sistema. Con una amplia gama de funciones y ejemplos prácticos, este manual permite a los usuarios aprender a programar de manera eficiente y efectiva, facilitando la implementación y optimización de procesos. Al proporcionar una documentación clara y concisa, el manual ayuda a los usuarios a aprovechar al máximo todas las capacidades del sistema, lo que resulta en una mayor productividad y éxito en sus proyectos. En resumen, el manual de programación es una herramienta esencial que empodera a los usuarios para alcanzar sus objetivos de manera efectiva y eficiente.