|  |  |
| --- | --- |
| Método de control de calidad de piezas. | Descripción breve  Sistema embebido en tiempo real para el control de calidad de piezas, utiliza un microcontrolador para evaluar las dimensiones y peso de piezas en una línea de producción.  Jesus Sedeño Perdigones  Sistemas empotrados en tiempo real. |

ÍNDICE DE CONTENIDO

Contenido

[1. Introducción 2](#_Toc189297745)

[2. Estado del arte. 2](#_Toc189297746)

[4. Objetivo del Sistema 4](#_Toc189297747)

[4.1 Requisitos del Sistema 4](#_Toc189297748)

[4.2 Sensores Utilizados 4](#_Toc189297749)

[4.3 Sensores de Peso: 5](#_Toc189297750)

[4.4 Pruebas de Aceptación 6](#_Toc189297751)

[4.5 Pruebas de Rendimiento en Producción 6](#_Toc189297752)

[5. Diseño del Sistema 6](#_Toc189297753)

[5.1 Diagrama de Descomposición Modular 6](#_Toc189297754)

[5.2 Relación entre los Módulos 7](#_Toc189297755)

[6. Explicación Ampliada del Sistema Embebido en Tiempo Real para el Control de Calidad de Piezas 8](#_Toc189297756)

[6. Funcionamiento del código. 11](#_Toc189297757)

[1. Configuración inicial 12](#_Toc189297758)

[2. Funciones principales 12](#_Toc189297759)

[3. Configuración del sistema 13](#_Toc189297760)

[4. Comunicación y sincronización 13](#_Toc189297761)

[5. Ciclo de funcionamiento 14](#_Toc189297762)

[6. Concurrencia y robustez 14](#_Toc189297763)

[7. Conclusión final 14](#_Toc189297764)

[6. El Código. 16](#_Toc189297765)

MEMORIA

# Introducción

Este trabajo tiene como objetivo investigar e implementar un sistema embebido en tiempo real para el control de calidad de piezas. Este sistema utiliza un microcontrolador para evaluar las dimensiones y peso de piezas en una línea de producción. Emplea sensores para medir las dimensiones y el peso, junto con un actuador para rechazar las piezas defectuosas.

Este ejemplo utiliza FreeRTOS para manejar tareas concurrentes y simula las mediciones. En un hardware real, deberías conectar sensores de peso y tamaño compatibles.

Toda la información se encuentra subida a la dirección github: [RobinAlestis/Sistema-Control-Calidad](https://github.com/RobinAlestis/Sistema-Control-Calidad)

# Estado del arte.

**¿Qué son los sistemas empotrados en tiempo real?**

Los sistemas empotrados son aquellos en los que un computador forma parte de un elemento mayor y tienen unas necesidades tecnológicas diferentes a los de las aplicaciones que se ejecutan en sistemas normales, como un portátil. Al formar parte de un sistema mayor, el computador debe conectarse con él y ejecutar los programas que tiene en su interior a un ritmo acompasado con la evolución de ese entorno.

**Retos tecnológicos**

Esto implica dos retos a la hora de su construcción. **Uno es el tiempo real**, como tenemos que acompasar la velocidad de funcionamiento de los programas a la del entorno, a veces hay que hacer más lenta o rápida la ejecución, con unas instrucciones de retraso o con la planificación. **El otro reto** es la concurrencia, que es la capacidad de hacer muchas cosas a la vez. Los sistemas son ricos y tienen muchas partes que deben ser controladas al mismo tiempo.

En el computador, lo que da esta capacidad de hacer varias cosas a la vez es el sistema operativo, un programa que gestiona todo el funcionamiento de los recursos; en realidad cada procesador solo puede hacer una cosa a la vez, pero el sistema va cambiando tan rápido de una a otra que crea la ilusión de que sí lo hace al mismo tiempo.

**¿Qué conseguimos con el pesado en un sistema de control de calidad?**

Cuando la totalidad y la integridad son importantes, puede comprobarse la calidad del producto y pueden recopilarse datos de calidad mediante un control de calidad basado en el peso. Los instrumentos de pesaje que cuentan con funciones nuevas e innovadoras proporcionan resultados precisos y trazables en unos segundos. Las aplicaciones avanzadas de la pantalla facilitan la tarea de los operarios a la hora de recopilar los datos necesarios a partir del proceso objetivo de pesaje y garantizan un funcionamiento rápido. Las funciones de la base de datos incorporada permiten recopilar datos fiables y completos del control de calidad.

Juntas, estas habilidades hacen que el control de calidad basado en el peso sea un modo asequible, fácil de implementar y seguro para comprobar la calidad, la integridad y la totalidad del producto.

**Comprobación de la integridad y mejora del rendimiento**, existen muchísimas preocupaciones a la hora de la fabricación que pueden abordarse con el control de calidad basado en el peso. Los parámetros se pueden verificar mediante ventanas de tolerancia personalizadas para garantizar que el número de piezas sea el correcto y para comprobar la integridad de los kits.

Cuando se trata de elementos fabricados, otra de las formas en las que el control de calidad basado en el peso puede ayudar es a través del pesaje de las carcasas, para asegurar que no haya huecos ni burbujas que puedan afectar al rendimiento. Se puede comprobar la aplicación y el grosor de los revestimientos.

**Mejora de la precisión y resultados inmediatos**, el control de calidad basado en el peso tiene muchas ventajas. Entre ellas, se incluyen los aspectos de la medición objetiva para lograr precisión, la captación de datos instantánea y los resultados inmediatos. Todo ello contribuye a mejorar la productividad, la facilidad de uso, la capacidad para realizar calibraciones según los estándares, la velocidad de captación de datos y la posibilidad de adaptar la aplicación a las necesidades del proceso.

**¿Qué conseguimos con la medición en un sistema de control de calidad?**

Los sensores de dimensiones son dispositivos utilizados para medir y evaluar el tamaño, forma, posición y orientación de piezas u objetos. Estos sensores se emplean en procesos de fabricación y control de calidad para garantizar que los productos cumplan con las especificaciones dimensionales requeridas.

# Objetivo del Sistema

El sistema de control de calidad de piezas tiene como finalidad la inspección automatizada y en tiempo real de los componentes producidos en una línea de fabricación. Este sistema está diseñado para evaluar la conformidad de cada pieza en función de sus dimensiones y peso, garantizando que solo aquellas que cumplen con los estándares predefinidos sean aprobadas. Las piezas defectuosas son rechazadas mediante mecanismos automatizados, minimizando la intervención humana y optimizando el rendimiento del proceso industrial.

### Requisitos del Sistema

* **Capacidad de Producción:** El sistema debe procesar un mínimo de **48 piezas por minuto**, asegurando una operación fluida sin afectar la eficiencia de la línea de producción.
* **Tolerancias Dimensionales y de Peso:**

**Dimensiones:** Longitud permitida:  **95 mm ± 105 mm**

**Peso:** Rango permitido: **500 g ± 600 g**

* **Precisión y Margen de Error**

**Precisión de Medición:** (±0.01 mm - ±0.1 mm) para sensores de dimensión, (±0.1 g - ±1 g) para sensores de peso.

* **Tiempos de Procesamiento**

**Tiempo máximo permitido para la evaluación de cada pieza:** 1 segundos.

**Tiempo de respuesta del actuador de rechazo:** Menos de 500 milisegundos.

### Sensores Utilizados

**Sensores de Dimensiones: Barreras de Luz (Fotodiodos y LED)**

* Consisten en un emisor (LED) y un receptor (fotodiodo o fototransistor) colocados en lados opuestos o en la misma estructura.
* El objeto interrumpe o modifica el paso del haz de luz, lo que permite medir dimensiones específicas como longitud, ancho o grosor.

**OEEB133U0135** Barrera para tareas de medición

**Texto

Descripción generada automáticamente**

**Interfaz de usuario gráfica, Texto

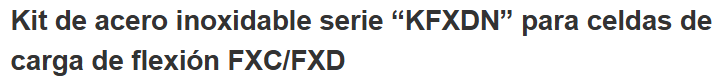
Descripción generada automáticamente**

**Ventajas:**

* Simplicidad y rapidez en la detección.
* Alta precisión en medidas lineales y en detección de bordes.
* Adecuados para piezas pequeñas y repetitivas.

### Sensores de Peso:

* Celdas de carga con precisiones de hasta 6000 g.



### Pruebas de Aceptación

Para garantizar la fiabilidad del sistema, se aplicarán las siguientes pruebas:

**Validación de Precisión de Sensores**

* + Comparación de mediciones obtenidas con patrones de referencia certificados.
  + Calibración de sensores para garantizar desviaciones mínimas.

**Prueba de Rechazo de Piezas Defectuosas**

* + Introducción controlada de piezas fuera de especificación para verificar la respuesta del sistema.
  + Medición de tasa de detección efectiva de defectos.

### Pruebas de Rendimiento en Producción

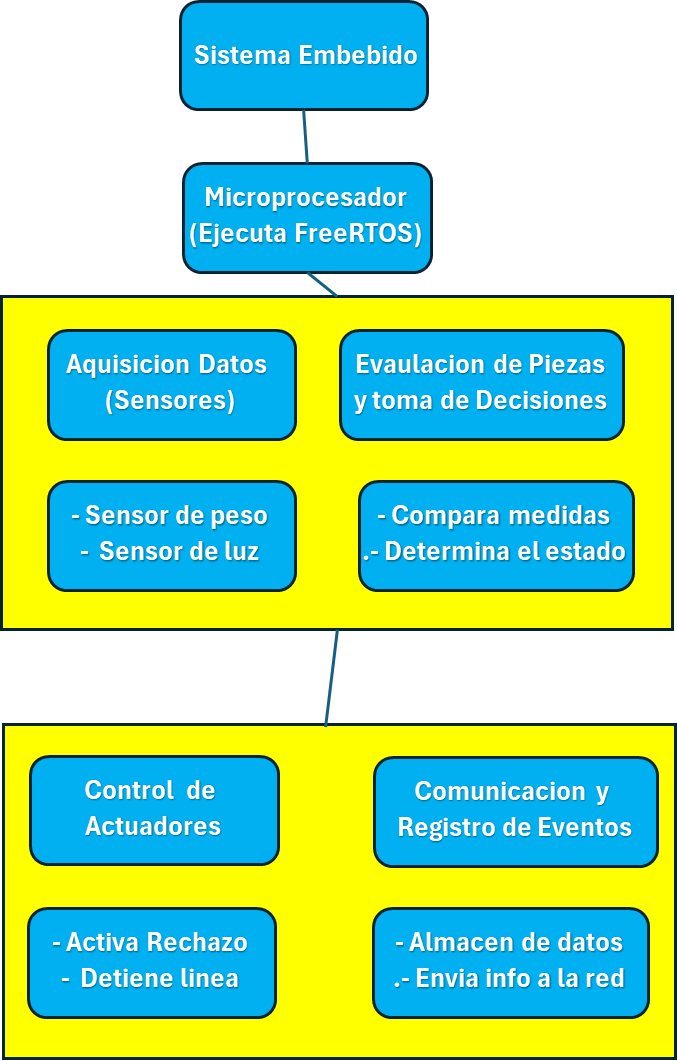
* + Evaluación del sistema en condiciones de carga máxima.
  + Medición de la latencia en la detección y actuación.

# Diseño del Sistema

Este sistema embebido en tiempo real permite evaluar la calidad de las piezas en una línea de producción a través de sensores de peso y dimensiones. Se organiza en varios módulos interconectados para garantizar un funcionamiento eficiente.

### Diagrama de Descomposición Modular

El sistema se compone de los siguientes módulos principales:



### Relación entre los Módulos

Cada módulo se comunica mediante colas y semáforos para garantizar la sincronización y evitar bloqueos.

1. **Módulo de Adquisición de Datos**
   * Captura la información de los sensores.
   * Convierte los datos en señales digitales para procesarlas.
   * Envío de datos al módulo de evaluación.
2. **Módulo de Evaluación de Piezas y Toma de Decisiones**
   * Compara los valores de peso y dimensión con los rangos permitidos.
   * Si la pieza es defectuosa, envía señal al módulo de actuadores.
3. **Módulo de Control de Actuadores**
   * Si recibe una señal de rechazo, activa un mecanismo para retirar la pieza defectuosa.
   * Puede detener la línea de producción si se detectan muchas piezas defectuosas.
4. **Módulo de Comunicación y Registro de Eventos**
   * Registra datos de las piezas evaluadas.
   * Puede enviar la información a una base de datos o interfaz de monitoreo.

# Explicación Ampliada del Sistema Embebido en Tiempo Real para el Control de Calidad de Piezas

Un **sistema embebido en tiempo real** diseñado para el control de calidad en una línea de producción se enfoca en evaluar las piezas fabricadas de forma automática y en tiempo real. Este tipo de sistemas permite identificar piezas defectuosas según parámetros predefinidos (como dimensiones, peso o forma) y retirarlas del flujo de producción para garantizar que solo los productos que cumplen con las especificaciones lleguen al siguiente proceso.

**Componentes del Sistema**

El sistema incluye varios módulos que interactúan para realizar las tareas de evaluación, decisión y acción:

1. **Microcontrolador:**
   * Es el "cerebro" del sistema embebido.
   * Ejecuta el software (basado en un sistema operativo en tiempo real como FreeRTOS).
   * Gestiona la captura de datos de los sensores y controla los actuadores.
2. **Sensores:**
   * **Sensores de peso:** Detectan el peso de cada pieza en tiempo real y envían los datos al microcontrolador.
   * **Sensores de dimensiones:** Evalúan el tamaño de las piezas, utilizando métodos como:
     + Barreras de luz (fotodiodos y LED).

Los sensores generan señales analógicas o digitales que son procesadas por el microcontrolador.

1. **Actuadores:**
   * Estos dispositivos realizan acciones físicas, como:
     + Desviar piezas defectuosas a un canal de rechazo.
     + Detener temporalmente la línea si hay demasiadas fallas consecutivas.
   * Los actuadores son controlados por el microcontrolador según los datos procesados.
2. **Sistema Operativo en Tiempo Real (RTOS):**
   * **FreeRTOS:** Administra múltiples tareas concurrentes, como:
     + Adquisición de datos de sensores.
     + Comparación de medidas con los valores aceptables.
     + Activación de actuadores en caso de fallo.
     + Registro de datos para auditoría o análisis.
3. **Interfaces de Comunicación:**
   * Permiten conectar el sistema a dispositivos externos para monitoreo o configuración:
     + Comunicación serial (UART, SPI, I2C).
     + Interfaces de red (Ethernet, Wi-Fi) para enviar datos a sistemas de control centralizados o en la nube.
4. **Flujo de Operación del Sistema:**
5. **Adquisición de Datos:**
   * La pieza pasa por la estación de evaluación, donde los sensores miden el peso y las dimensiones.
   * Los datos son enviados al microcontrolador, que los convierte a valores digitales si es necesario (mediante un ADC).
6. **Procesamiento:**
   * El microcontrolador ejecuta un algoritmo que compara las medidas de cada pieza con los valores límites establecidos (por ejemplo, dimensiones mínimas/máximas y peso tolerado).
   * Este procesamiento ocurre en tiempo real, asegurando que la línea de producción no se detenga.
7. **Toma de Decisiones:**
   * Si la pieza cumple con los parámetros, se permite que siga en la línea de producción.
   * Si no cumple, se activa el actuador correspondiente para rechazar la pieza (p. ej., una compuerta neumática o un brazo mecánico).
8. **Registro de Eventos:**
   * Cada decisión es registrada en un sistema de almacenamiento local o enviado a un servidor externo para análisis posterior.
9. **FreeRTOS y la Gestión de Tareas Concurrentes**

En este sistema, FreeRTOS es clave para manejar las operaciones concurrentes, divididas en tareas:

* **Tarea de Adquisición de Datos:**
  + Lee los valores de los sensores a intervalos regulares.
  + Preprocesa los datos (filtrado de ruido, conversión de unidades).
  + En hardware real, utiliza interrupciones para reaccionar rápidamente a las señales.
* **Tarea de Evaluación:**
  + Compara los valores de peso y dimensiones con los parámetros predefinidos.
  + Decide si la pieza es aceptable o defectuosa.
* **Tarea de Control de Actuadores:**
  + Activa los actuadores para desviar piezas defectuosas.
  + Sincroniza el movimiento del actuador con el flujo de la línea de producción.
* **Tarea de Registro:**
  + Guarda datos sobre el estado de la línea (número de piezas aceptadas/rechazadas, tiempos de ciclo, etc.).
  + Transmite datos a sistemas externos si es necesario.
* **Gestión del Tiempo:**
  + Las tareas tienen prioridades definidas; por ejemplo:
    - Adquisición de datos tiene prioridad alta, ya que debe ejecutarse en intervalos precisos.
    - El registro de eventos puede tener prioridad baja, ya que no es crítico para el funcionamiento inmediato.

1. **Simulación del Sistema**

En un entorno simulado, como el que describe el ejemplo:

1. **Sensores Simulados:**
   * Se generan valores aleatorios dentro de un rango definido para representar las mediciones.
   * Esto permite probar el sistema antes de conectarlo a hardware real.
2. **Actuadores Simulados:**
   * En lugar de accionar un dispositivo físico, se registra el evento (p. ej., "pieza rechazada").
3. **Pruebas y Verificación:**
   * El sistema puede ser evaluado en condiciones controladas para garantizar que responde correctamente a diferentes escenarios (piezas válidas, defectuosas, etc.).
4. **Hardware Real:**

En un entorno de producción real, el sistema utiliza sensores y actuadores físicos:

* **Sensores:** Básculas industriales, escáneres láser o fotodiodos.
* **Actuadores:** Sistemas neumáticos, motores eléctricos o solenoides.

El diseño embebido debe incluir interfaces para integrar estos componentes y garantizar que las decisiones se ejecuten en tiempo real.

**Conclusión**

Este sistema embebido en tiempo real es un ejemplo claro de cómo los sistemas modernos automatizan procesos industriales. Usando FreeRTOS, sensores avanzados y actuadores confiables, el sistema logra:

* **Eficiencia:** Procesa piezas a alta velocidad.
* **Fiabilidad:** Detecta defectos consistentemente.
* **Escalabilidad:** Puede adaptarse para incluir más parámetros de calidad o integrarse con sistemas de monitoreo centralizados.

Estos sistemas son fundamentales para garantizar calidad y competitividad en la producción industrial moderna.

# Funcionamiento del código.

El código está escrito en C y está diseñado para ser ejecutado en un entorno basado en FreeRTOS, como un microcontrolador ESP32. El propósito principal del programa es simular un sistema de control de calidad para una línea de producción, donde se verifican las dimensiones y el peso de piezas usando sensores, y se toman decisiones sobre si aceptar o rechazar cada pieza en función de umbrales definidos. La explicación por cada parte:

### Configuración inicial

* **Definición de pines y parámetros:**
  + **SENSOR\_DIMENSION\_PIN y SENSOR\_WEIGHT\_PIN** representan los pines de entrada para sensores que miden la dimensión y el peso de las piezas.
  + **ACTUATOR\_REJECT\_PIN** es el pin de salida para un actuador que rechaza piezas defectuosas.
  + Constantes como **DIMENSION\_MIN, DIMENSION\_MAX, WEIGHT\_MIN, y WEIGHT\_MAX** definen los umbrales aceptables para las dimensiones y el peso de las piezas.
* **Buffers y comunicación:**
  + **test\_data:** un buffer cíclico con capacidad para 5 elementos para almacenar datos generados por los sensores.
  + **quality\_check\_queue:** una cola para transferir datos entre tareas.
* **Semáforos:**
  + **buffer\_semaphore:** controla el acceso concurrente al buffer test\_data.
  + **count\_semaphore:** protege las variables compartidas accepted\_count y rejected\_count.

### Funciones principales

* **generate\_test\_data**

Genera datos aleatorios que simulan lecturas de los sensores.

1. Cada segundo, intenta generar un nuevo dato con un 80% de probabilidad (basado en **SKIP\_PROBABILITY**).
2. Si genera un dato:

* Se asegura de que el buffer no esté lleno usando **buffer\_semaphore.**
* Añade el nuevo dato al buffer cíclico **test\_data.**
* Actualiza los índices y el contador del buffer.

1. Si el buffer está lleno, el dato no se añade y se registra un mensaje.

* **sensor\_task**

Lee todos los datos almacenados en **test\_data** y los envía a la cola **quality\_check\_queue.**

1. Usa **buffer\_semaphore** para acceder al buffer **test\_data** de forma segura.
2. Si hay datos en el buffer:

* Lee y elimina cada dato del buffer.
* Envía cada dato a **quality\_check\_queue** usando **xQueueSend.**

1. Si el buffer está vacío, muestra un mensaje indicando que no hay datos.

* **quality\_check\_task**

Recibe datos desde la cola **quality\_check\_queue** y los analiza para determinar si cumplen con los requisitos de calidad.

1. Bloquea la tarea hasta que reciba datos desde la cola.
2. Verifica si el dato cumple con los umbrales de dimensiones y peso.

* Si el dato es aceptable, incrementa **accepted\_count.**
* Si el dato es defectuoso:
  + - Incrementa **rejected\_count.**
    - Activa el actuador de rechazo durante 0.5 segundos.

1. Imprime el resultado del análisis y los conteos actualizados de piezas aceptadas y rechazadas.

### Configuración del sistema

* **app\_main**

Función principal que configura y arranca el sistema.

1. Configura el pin para el actuador usando la función **gpio\_config.**
2. Crea:
   * La cola **quality\_check\_queue** para comunicación entre tareas.
   * Los semáforos **buffer\_semaphore** y **count\_semaphore** para sincronización.
3. Crea tres tareas concurrentes:
   * **generate\_test\_data** para simular datos del sensor.
   * **sensor\_task** para transferir datos del buffer a la cola.
   * **quality\_check\_task** para analizar los datos de calidad.
4. Inicia el sistema con un mensaje en la consola.

### Comunicación y sincronización

* Cola **quality\_check\_queue:**
* Actúa como medio de comunicación entre **sensor\_task** y **quality\_check\_task.**
* Permite pasar datos de una tarea a otra de manera segura.
* Semáforos:
* **buffer\_semaphore:** Protege el acceso al buffer cíclico test\_data, asegurando que solo una tarea acceda o modifique el buffer al mismo tiempo.
* **count\_semaphore:** Protege las variables compartidas accepted\_count y rejected\_count para evitar condiciones de carrera.

### Ciclo de funcionamiento

1. Generación de datos:

* **generate\_test\_data** crea datos simulados y los almacena en el buffer **test\_data.**

1. Procesamiento de datos:

* **sensor\_task** lee los datos del buffer y los transfiere a **quality\_check\_queue.**

1. Análisis de datos:

* **quality\_check\_task** recibe los datos desde la cola, verifica su calidad, y actualiza los contadores.

1. Rechazo de piezas:

* Si una pieza no cumple los requisitos, el actuador se activa brevemente para rechazarla.

### Concurrencia y robustez

* Protección contra accesos concurrentes: Los semáforos aseguran la consistencia de los datos compartidos entre tareas.
* Tolerancia al llenado del buffer: Si el buffer **test\_data** se llena, **generate\_test\_data** descarta los nuevos datos, evitando bloqueos.
* Cola de procesamiento: Garantiza que los datos se procesen en orden y evita que se pierdan durante el análisis.

Este diseño asegura un flujo ordenado y confiable en el control de calidad de las piezas.

# Conclusión final

Este código implementa un **sistema básico de control de calidad en una línea de producción** utilizando FreeRTOS. La estructura refleja un diseño concurrente eficiente donde las tareas trabajan en conjunto para simular un flujo de datos real, procesarlos y tomar decisiones basadas en criterios predefinidos. A continuación, destacamos los puntos clave:

1. **Modularidad y claridad:**
   * El código está dividido en tareas bien definidas (generate\_test\_data, sensor\_task, quality\_check\_task), lo que facilita la comprensión, mantenimiento y ampliación del sistema.
2. **Uso de recursos de FreeRTOS:**
   * La cola (xQueueCreate) permite una comunicación fluida entre tareas sin conflictos de acceso.
   * Los semáforos (xSemaphoreCreateMutex) garantizan la sincronización de recursos compartidos, evitando condiciones de carrera y asegurando la integridad de los datos.
3. **Manejo de recursos limitados:**
   * El buffer cíclico (FIFO) limita el almacenamiento de datos para evitar sobrecarga. La lógica implementada maneja con gracia los casos en los que el buffer está lleno.
4. **Tolerancia y robustez:**
   * El sistema está diseñado para operar de manera robusta en un entorno de datos variables, descartando algunos datos (simulando ruido del sensor) y rechazando piezas defectuosas.
5. **Simulación de un entorno real:**
   * Los datos generados aleatoriamente simulan condiciones de una fábrica, lo que hace que el código sea ideal como una práctica introductoria para sistemas embebidos en la industria.
6. **Capacidad de expansión:**
   * El diseño puede ampliarse fácilmente, por ejemplo, añadiendo más sensores, actuadores, o nuevas reglas de calidad.

En resumen, este ejercicio muestra cómo integrar conceptos clave de sistemas embebidos y tiempo real (sincronización, comunicación entre tareas y control de dispositivos) para construir una solución funcional y escalable en un entorno de producción industrial. Es una excelente base para desarrollar aplicaciones más complejas en el campo del IoT o la automatización industrial.

# El Código.

El presente código implementa un flujo automatizado para evaluar piezas según dimensiones y peso, utilizando tareas concurrentes, colas para comunicación, y semáforos para sincronización. Además, incluye la gestión de un buffer cíclico y el control de un actuador para rechazar piezas fuera de especificaciones.

#include <stdio.h>

#include <stdlib.h>

#include "freertos/FreeRTOS.h"

#include "freertos/task.h"

#include "freertos/queue.h"

#include "freertos/semphr.h"

#include "driver/gpio.h"

// Definición de pines (ajustar según hardware real)

#define SENSOR\_DIMENSION\_PIN GPIO\_NUM\_4

#define SENSOR\_WEIGHT\_PIN GPIO\_NUM\_5

#define ACTUATOR\_REJECT\_PIN GPIO\_NUM\_18

// Umbrales para control de calidad

#define DIMENSION\_MIN 95 // Dimensión mínima en mm

#define DIMENSION\_MAX 105 // Dimensión máxima en mm

#define WEIGHT\_MIN 500 // Peso mínimo en gramos

#define WEIGHT\_MAX 600 // Peso máximo en gramos

// Probabilidad de no añadir datos nuevos (20%)

#define SKIP\_PROBABILITY 20

// Tamaño del buffer cíclico

#define BUFFER\_SIZE 5

// Cola para comunicación entre tareas

static QueueHandle\_t quality\_check\_queue = NULL;

// Semáforos para protección

static SemaphoreHandle\_t buffer\_semaphore = NULL;

static SemaphoreHandle\_t count\_semaphore = NULL;

// Variables para conteo de piezas

static volatile int accepted\_count = 0;

static volatile int rejected\_count = 0;

// Estructura para pasar los datos

typedef struct {

int dimension;

int weight;

} piece\_data\_t;

// Buffer cíclico para almacenar datos generados

static piece\_data\_t test\_data[BUFFER\_SIZE];

static int buffer\_start = 0;

static int buffer\_end = 0;

static int buffer\_count = 0;

// Función para generar un dato nuevo aleatorio

void generate\_test\_data(void \*pvParameters) {

while (1) {

// Intentar generar datos aleatoriamente con una probabilidad de SKIP\_PROBABILITY

if ((rand() % 100) < SKIP\_PROBABILITY) {

printf("generate\_test\_data: No se generaron nuevos datos en este ciclo.\n");

vTaskDelay(pdMS\_TO\_TICKS(1000)); // Esperar antes del próximo ciclo

continue;

}

// Generar datos aleatorios dentro de los rangos especificados

piece\_data\_t new\_data = {

.dimension = 90 + rand() % 21, // Entre 90 y 110 mm

.weight = 480 + rand() % 141 // Entre 480 y 620 g

};

// Proteger acceso al buffer cíclico con un semáforo

xSemaphoreTake(buffer\_semaphore, portMAX\_DELAY);

if (buffer\_count < BUFFER\_SIZE) {

// Añadir datos al buffer cíclico

test\_data[buffer\_end] = new\_data;

buffer\_end = (buffer\_end + 1) % BUFFER\_SIZE;

buffer\_count++;

printf("generate\_test\_data: Dato generado - Dimensión: %d mm, Peso: %d g\n", new\_data.dimension, new\_data.weight);

} else {

printf("generate\_test\_data: Buffer lleno, no se añadieron nuevos datos.\n");

}

xSemaphoreGive(buffer\_semaphore);

vTaskDelay(pdMS\_TO\_TICKS(1000)); // Esperar antes del próximo ciclo

}

}

// Tarea para leer todos los datos del buffer y enviarlos a la cola

void sensor\_task(void \*pvParameters) {

while (1) {

xSemaphoreTake(buffer\_semaphore, portMAX\_DELAY);

if (buffer\_count > 0) {

printf("sensor\_task: Procesando %d datos del buffer.\n", buffer\_count);

while (buffer\_count > 0) {

// Leer datos del buffer cíclico

piece\_data\_t data = test\_data[buffer\_start];

buffer\_start = (buffer\_start + 1) % BUFFER\_SIZE;

buffer\_count--;

// Enviar datos a la cola

if (xQueueSend(quality\_check\_queue, &data, portMAX\_DELAY) != pdPASS) {

printf("sensor\_task: Error al enviar datos a la cola.\n");

}

}

printf("sensor\_task: Todos los datos procesados y enviados a la cola.\n");

} else {

printf("sensor\_task: No hay datos en el buffer para procesar.\n");

}

xSemaphoreGive(buffer\_semaphore);

vTaskDelay(pdMS\_TO\_TICKS(1000)); // Esperar antes del próximo ciclo

}

}

// Tarea para analizar los datos (control de calidad)

void quality\_check\_task(void \*pvParameters) {

piece\_data\_t data;

while (1) {

// Esperar datos de la cola (bloqueante)

if (xQueueReceive(quality\_check\_queue, &data, portMAX\_DELAY)) {

printf("Evaluando - Dimensión: %d, Peso: %d (Límites: %d-%d, %d-%d)\n",

data.dimension, data.weight, DIMENSION\_MIN, DIMENSION\_MAX, WEIGHT\_MIN, WEIGHT\_MAX);

// Comprobar si la pieza cumple los requisitos

if (data.dimension >= DIMENSION\_MIN && data.dimension <= DIMENSION\_MAX &&

data.weight >= WEIGHT\_MIN && data.weight <= WEIGHT\_MAX) {

xSemaphoreTake(count\_semaphore, portMAX\_DELAY);

accepted\_count++;

xSemaphoreGive(count\_semaphore);

printf("Pieza ACEPTADA - Dimensión: %d mm, Peso: %d g\n", data.dimension, data.weight);

} else {

xSemaphoreTake(count\_semaphore, portMAX\_DELAY);

rejected\_count++;

xSemaphoreGive(count\_semaphore);

printf("Pieza RECHAZADA - Dimensión: %d mm, Peso: %d g\n", data.dimension, data.weight);

// Activar actuador para rechazar la pieza

gpio\_set\_level(ACTUATOR\_REJECT\_PIN, 1);

vTaskDelay(pdMS\_TO\_TICKS(500)); // Activar actuador por 0.5 segundos

gpio\_set\_level(ACTUATOR\_REJECT\_PIN, 0);

}

// Imprimir conteo de piezas buenas y malas

printf("Piezas aceptadas: %d, Piezas rechazadas: %d\n", accepted\_count, rejected\_count);

}

}

}

// Configuración inicial del hardware

void app\_main(void) {

// Configuración del pin del actuador

gpio\_config\_t io\_conf = {

.pin\_bit\_mask = (1ULL << ACTUATOR\_REJECT\_PIN),

.mode = GPIO\_MODE\_OUTPUT,

.pull\_up\_en = GPIO\_PULLUP\_DISABLE,

.pull\_down\_en = GPIO\_PULLDOWN\_DISABLE,

.intr\_type = GPIO\_INTR\_DISABLE

};

gpio\_config(&io\_conf);

// Crear la cola para la comunicación

quality\_check\_queue = xQueueCreate(10, sizeof(piece\_data\_t));

if (quality\_check\_queue == NULL) {

printf("Error: No se pudo crear la cola.\n");

return;

}

// Crear semáforos

buffer\_semaphore = xSemaphoreCreateMutex();

count\_semaphore = xSemaphoreCreateMutex();

if (buffer\_semaphore == NULL || count\_semaphore == NULL) {

printf("Error: No se pudieron crear los semáforos.\n");

return;

}

// Crear tareas

xTaskCreate(generate\_test\_data, "generate\_test\_data", 2048, NULL, 5, NULL);

xTaskCreate(sensor\_task, "sensor\_task", 2048, NULL, 5, NULL);

xTaskCreate(quality\_check\_task, "quality\_check\_task", 2048, NULL, 5, NULL);

printf("Sistema de control de calidad iniciado.\n");

}