Implementación de STM32 en Sistemas de Control de Motores

Fernando Cuéllar Martínez, Jorge Antonio Villegas Hernández, María Fernanda Hernández Montes, Melissa Montemayor Riojas, Melissa Vélez Martínez[a]

[a]*Instituto Tecnológico y de Estudios Superiores de Monterrey,*

Abstract

En la actualidad nos encontramos ante un entorno en el que resulta fundamental la implementación de sistemas de control dentro de las aplicaciones robóticas, especialmente considerando el creciente interés en proyectos tales como la navegación de vehículos inteligentes. Dentro de este marco, los sistemas de control de lazo cerrado son una parte vital de la automatización, pues permiten monitorear y ajustar constantemente el comportamiento de un proceso o sistema en tiempo real, haciendo uso de retroalimentación para mantener el sistema en un estado deseado. En el presente escrito se realizará la documentación del procedimiento seguido para lograr el desarrollo del control de un gripper con dos grados de libertad.

Keywords: Sistemas de control, lazo cerrado, STM32, vehículos autónomos, sistemas embebidos

1. Introducción

Con la finalidad de diseñar un sistema autónomo capaz de desplazarse sobre un entorno físico e interactuar con este por medio de un efector final, se ha implementado un gripper que mantiene conexión e intercambio de datos a través de un microcontrolador (MCU) STM32, lo que nos permitirá desarrollar un sistema de control que mantenga el comportamiento del gripper dentro de un rango deseado. La tarjeta de microcontrolador usada es la tarjeta STM32F411CEU6, basada en ARM Cortex-M4, con una velocidad de reloj de hasta 100MHz, misma que nos ayudará a controlar dos motores distintos. El objetivo final es permitir que el vehículo autónomo Puzzlebot desarrollado por nuestro equipo sea capaz de interactuar con un conjunto de objetos cúbicos. Dichos cubos serán parte importante de un circuito en el que el robot se desplazará para recolectarlos y deberá evadir obstáculos hasta llegar a las zonas de descarga en las que deberá soltarlos. La zona de descarga específica será determinada por un equipo de jurado, y para que el robot complete el reto deberá estar equipado con un gripper de dos grados de libertad; uno de los grados estará encargado de subir y bajar el gripper, mientras que el segundo nos permitirá abrir y cerrar las tenazas encargadas de manipular los objetos.

De igual manera se le ha brindado un enfoque de aplicación adicional al desarrollo previamente planteado, donde el sistema Puzzlebot actuará como sorteador de frutas configurado para identificar y desechar frutas en mal estado dado un lote de cosecha. Para el análisis de sus implicaciones se ha creado un mapa de impacto y contexto de uso, considerando al usuario, entorno local, servicio de suministros y sistema de soporte. Al considerar el entorno local, se deben tener en cuenta diversos factores. En primer lugar, la posición de los objetos que se deben clasificar es fundamental, además de que la iluminación del entorno puede influir en la visión y percepción de los colores, especialmente si la clasificación se basa en esta variable. Por otro lado, es crucial que el robot tenga la capacidad de evitar colisiones con obstáculos presentes en su entorno. En los suministros se considera la materia prima, contenedores para las frutas y fuente de poder para ejecutar el proceso con el robot. Para llevar un monitoreo y sistema de soporte, se considera el estatus actual del sistema, teniendo un sistema de retroalimentación para poder realizar actualizaciones en caso de ser necesario. Finalmente, el usuario que maneja el robot podrá confirmar el proceso de clasificación y selección, proveyéndole un manual de usuario. Este planteamiento puede ser observado en el diagrama 1.

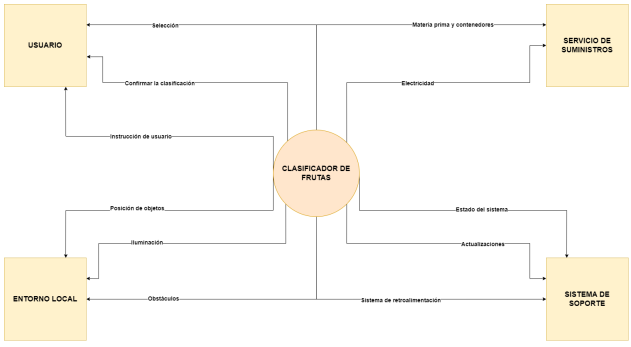

Figure 1: Mapa de impacto y contexto de uso.

2. Desarrollo

Ahora bien, la implementación de una tarjeta STM32 para controlar un sistema embebido de gripper ofrece la posibilidad de diseñar una solución eficiente y precisa para manipulación y agarre en diversos contextos. La tarjeta STM32, conocida por su potencia de procesamiento y flexibilidad, proporciona una base sólida para desarrollar un controlador robusto y de alto rendimiento.

Con el objetivo de controlar un sistema embebido de gripper, la tarjeta STM32 puede ser programada para interactuar con los sensores y actuadores necesarios para el funcionamiento del gripper. Mediante la configuración adecuada de los pines GPIO y la programación de los periféricos integrados, como los módulos de temporizadores y comunicación serial, es posible establecer la comunicación y el control preciso de los componentes del gripper.

La tarjeta STM32 también brinda la posibilidad de aprovechar las ventajas de su entorno de desarrollo, que incluye un amplio conjunto de herramientas y bibliotecas. Estas facilitan la implementación de algoritmos de control avanzados, como la cinemática inversa o el control de fuerza, lo que permite adaptar el gripper a diferentes aplicaciones y requisitos específicos.

Se ofrece una amplia gama de microcontroladores STM32 con diferentes capacidades de memoria, clock rate, periféricos y opciones de conectividad. Esto permite seleccionar la tarjeta más adecuada para las funciones que se planean ejecutar, ya sean aplicaciones de bajo consumo de energía, de alto rendimiento, etc.

El microcontrolador utilizado en este proyecto es el modelo STM32F411CEU6, con 32 bits basado en ARMCortex-M4 con una velocidad de reloj de hasta 100MHz. Cuenta con las siguientes especificaciones:

Aspecto	Descripción
Memoria flash	512 kB
Memoria RAM	128 kB
GPIO:	82
ADC	3 de 12 bits
Temporizadores	14
Comunicación serial	3 puertos USART, 2 puertos SPI, 3 puertos I2C
Comunicación USB	1 puerto USB 2.0
Periféricos adicionales	DMA, RTC, WDG, CRC1, RNG

Figure 2: Especificaciones de modelo STM32F411CEU6

Se optó por este modelo de STM principalmente por su potencia de procesamiento, pues permite una amplia capacidad de almacenamiento para el código del programa, también debido a su conectividad y eficiencia energética.

No obstante, si bien contamos con un contexto general óptimamente planteado, es necesario tomar en consideración una serie de requerimientos y limitantes dentro de nuestro desarrollo.

2.1. Requerimientos del Gripper y Control Manipulador

En seguida se enlista una serie de requerimientos tanto para el gripper como para el control manipulador.

Mecánica

El dispositivo debe de tener todas sus conexiones eléctricas correctamente enlazadas para evitar interferencias.

Hardware

Protección y diagnosis. El dispositivo debe ser capaz de identificar las siguientes condiciones de error:

- Cuando la posición angular indicada del manipulador y la posición angular del gripper no coinciden.
- El sensor del gripper no está detectando contacto para levantar la pieza.

El dispositivo debe detener operaciones cuando alguna de las condiciones de error se identifica.

Pinout y cableado.

- El dispositivo debe de tener una conexión destinada para la comunicación.
- El conector de pinout debe ser definido como puerto de comunicación para commando y debugging
- El dispositivo debe usar un pin destinado para la medición de la posición angular del manipulador.
- El dispositivo debe usar un pin destinado para la medición de la posición angular del gripper.
- El dispositivo debe usar un pin destinado para el sensor de contacto entre el objeto a levantar y el gipper.

Display. El sistema debe tener un display digital, una consola de salida o algún indicador visual para mostrar el estatus del subsistema del manipulador.

Arquitectura de Software

- Los datos se deben recibir y transmitir a través de una comunicación serial (SPI, I2C o Serial TTL).
- El dispositivo debe recibir comandos de posición sobre el puerto de comunicación serial para ejecutar los movimientos del manipulador y gripper.

Funcionamiento de Software

- El sistema deberá implementar un medio de verificación para permitir la detección de circuito abierto a través de un fusible de software.
- El software fuse debe ser implementado en una tarea que se ejecute cada 10ms.

2.2. Distribución de tareas

Al comprender los requisitos y funcionalidades requeridas para el sistema embebido de gripper, se pueden identificar los componentes críticos y las dependencias entre ellos. Con esta información, se puede diseñar una estructura de tareas que aproveche al máximo las capacidades de la tarjeta STM32 y optimice la utilización de los recursos disponibles.

Mediante una cuidadosa planificación y asignación de las tareas, se puede determinar qué módulos o funcionalidades se pueden desarrollar simultáneamente y cuáles requieren de una secuencia específica. Esto permite un enfoque de desarrollo paralelo, donde los miembros del equipo pueden trabajar en diferentes aspectos del proyecto al mismo tiempo, reduciendo así el tiempo total de desarrollo. Esta distribución de tareas puede ser observada en el diagrama 3

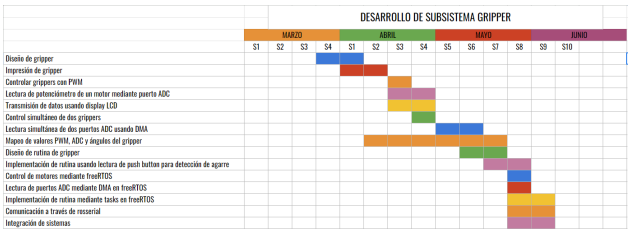

Figure 3: Distribución de tareas.

3. Rutinas diseñadas mediante el uso puro de STM32

Con el objetivo de facilitar el desarrollo del sistema embebido del gripper, se ha implementado el entorno STMCubeIDE. Esta herramienta, proporcionada por STMicroelectronics, brinda un entorno integrado de desarrollo (IDE) que permite generar rutinas y configuraciones para los microcontroladores STM32. La elección de STMCubeIDE se basó en su compatibilidad con la plataforma STM32 utilizada en el proyecto y su capacidad para simplificar la configuración y programación de los periféricos del microcontrolador.

Mediante el entorno STMCubeIDE, se pudo generar código inicial y configuraciones para los componentes del gripper, como sensores, actuadores y comunicaciones. La interfaz intuitiva y la amplia gama de opciones de configuración del entorno permitieron adaptar rápidamente los pines GPIO, configurar los modos de funcionamiento de los periféricos y generar código básico para interactuar con los componentes del sistema.

Esta implementación inicial en STMCubeIDE fue especialmente útil para plantear el funcionamiento y los casos de uso del gripper antes de su integración con el sistema operativo en tiempo real FreeRTOS. Al tener una base de código funcional y bien configurada, fue posible realizar pruebas y demostraciones de los movimientos y acciones del gripper de manera independiente antes de su incorporación al entorno más complejo de FreeRTOS.

En este contexto, a continuación se mencionan algunas de las configuraciones más relevantes para comprender la implementación y asignación de puertos en las diversas pruebas desarrolladas.

- ***Puertos ADC1*** Finalmente, este puerto se refiere al convertidor analógico-digital, mismo que nos ofrece, tal como lo dice su nombre, canales de entrada analógica para adquirir y convertir las señales. ADC1 puede funcionar en modo de conversión única, pero es el modo de conversión continua la que nos permite que la lectura realizada se actualice en tiempo real. De esta manera, los puertos son integrados con el fin de permitir la funcionalidad de nuestro sistema. Esto nos da paso a establecer un esquema de trabajo para posteriormente traducir el voltaje recibido en un ángulo de apertura, traducción a través de la cual en un futuro seremos capaces de alimentar el sistema de control tanto del motor 1 como del motor 2.
- ***Puertos TIM2*** El puerto TIM2 se refiere al Timer 2, que es un periférico de temporización, y dentro de esta aplicación en específico nos permite generar señales que son enviadas periódicamente al motor, lo que se traduce en el voltaje recibido para su funcionamiento.
- ***Puertos SYS*** En este caso, los puertos SYS utilizados para la interfaz de depuración y programación Serial Wire Debug (SWD) diseñada específicamente para microcontroladores ARM Cortex-M, como los STM32. El puerto etiquetado como SWDIO , generalmente funciona como señal bidireccional, de manera que funciona como entrada y salida de datos entre la máquina en la que se programa el funcionamiento del STM32 y la tarjeta en sí. Se usa para enviar comandos de depuración, lectura y escritura de registros internos del microcontrolador, además de que también nos permite acceder a la memoria del programa, así como monitorear y controlar el estado del microcontrolador durante la ejecución. Por otra parte, el puerto etiquetado como SWCLK, generalmente se configura como una señal de reloj que sincroniza la transferencia de datos entre el depurador o programador y el microcontrolador. Cuando se genera la depuración y programación, este pin funciona para proporcionar una señal de reloj estable y sincronizada, lo que permite coordinar el intercambio de datos y controlar el flujo de información y sincronización de las operaciones de lectura y escritura. La frecuencia de su señal se configura de acuerdo con los requisitos de velocidad de comunicación y las limitaciones propias del microcontrolador, lo que quiere decir que la velocidad de reloj debe ser lo suficientemente alta para permitir una transferencia de datos eficiente, pero también debe ser compatible con las

especificaciones de los dispositivos involucrados.

- ***Puertos RCC*** Reset and Clock Control (RCC) es el módulo encargado de la configuración del reloj y periféricos del sistema. Inicializar estos puertos nos permite seleccionar la fuente de reloj del microcontrolador, lo que establece la frecuencia del sistema, además de que también sirve para habilitar o deshabilitar los relojes de los periféricos individuales según sea necesario ahorrar energía y optimizar el rendimiento del sistema.

- ***Puertos USART (PA9 y PA10)*** USART es el acrónimo de Universal Synchronous/Asynchronous Receiver/Transmitter, que es un periférico para comunicación serial. Nos proporciona capacidades de transmisión y recepción de datos en modo síncrono y asíncrono, lo que nos permite establecer comunicación con otros periféricos, como es el caso de nuestro motor. En este caso, USART1 se activa con la finalidad de permitir la comunicación síncrona con el puerto ADC, de manera que, una vez que los datos sean recibidos, puedan ser almacenados en el registro de datos para que el programa los lea y utilice para nuestras futuras aplicaciones.

Previo al desarrollo de una rutina que nos permitiera comprender los casos de uso del sistema gripper ha resultado de especial utilidad la implementación de pruebas unitarias que nos brindaran las bases más relevantes para la adquisición de un conjunto de habilidades necesarias dentro de la comprensión tanto del entorno de desarrollo como de las implicaciones técnicas de su implementación física. Las fichas técnicas de las mismas pueden ser consultadas en la sección 8.

Como se mencionó anteriormente, las pruebas unitarias fueron de gran utilidad para la obtención de una rutina que nos permitiera plantear los casos de uso del gripper en cada uno de sus estados, lo que se representa gráficamente en la figura 4.

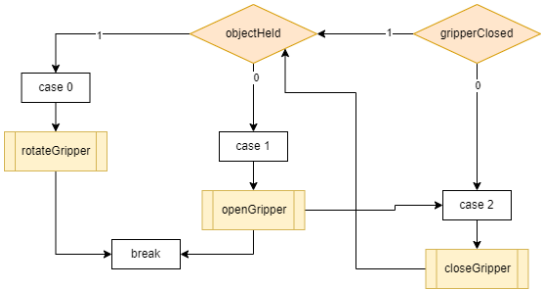

Figure 4: Casos de uso para gripper.

Enlace al codigo implementado para casos de uso: https://github.com/Albatambien/controlGripperSTM32 /tree/-master/servomotores_Pruebas

4. Diseño de MBSD para el filtrado de las señales

El diseño de Model-Based System Design (MBSD) desempeña un papel fundamental en el filtrado de señales utilizando la herramienta Simulink. El filtrado de señales es esencial en numerosas aplicaciones, desde procesamiento de audio hasta procesamiento de imágenes y señales biomédicas. Mediante el enfoque de MBSD, es posible diseñar sistemas de filtrado eficientes y optimizados, lo que permite obtener resultados precisos y confiables.

La herramienta Simulink proporciona un entorno de diseño y simulación poderoso y flexible para el desarrollo de sistemas basados en modelos. Al aprovechar las capacidades de Simulink, los ingenieros pueden diseñar y validar modelos de filtros de señales de manera intuitiva y visual. Esto permite una comprensión clara de las características del sistema y de cómo interactúan los diferentes componentes del filtro.

El diseño de MBSD en Simulink también ofrece la posibilidad de realizar simulaciones detalladas para evaluar el desempeño del filtro en diferentes escenarios y condiciones. Esto facilita la identificación y corrección de posibles problemas o limitaciones en el diseño antes de su implementación práctica. Además, Simulink permite la optimización de los parámetros del filtro, lo que contribuye a obtener un rendimiento óptimo y una respuesta deseada en la señal filtrada.

Como podemos observar en la figura 5, se implementó un sistema de control haciendo uno de Simulink con la finalidad de contar con una serie de parámetros en los que se basa posteriormente nuestro diseño de rutina utilizando la herramienta FreeRTOS.

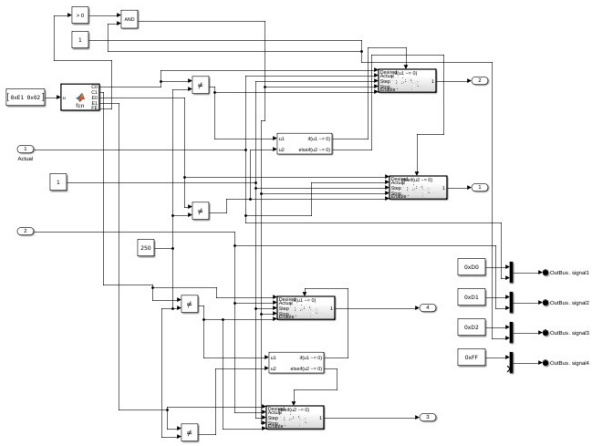

Figure 5: MBSD en Simulink

5. Implementación de FreeRTOS

Como sabemos, la implementación de sistemas embebidos nos permite optimizar la utilización de recursos para el desarrollo de tareas específicas y preestablecidas. No obstante, esto no quiere decir que por sí mismos no requieran igualmente de una administración eficiente de recursos y tareas para garantizar su correcto funcionamiento. En este contexto, FreeRTOS se destaca como un sistema operativo en tiempo real de código abierto, diseñado especialmente para aplicaciones de sistemas embebidos como la que nos encontramos desarrollando con la tarjeta de distribución STM32.

FreeRTOS fue desarrollado por Richard Barry y su objetivo es brindar una solución ligera y de alto rendimiento para sistemas embebidos con restricciones de recursos. Se encuentra basado en un modelo de programación de tareas y ofrece una amplia gama de funcionalidades, por lo que puede ser interpretado como una librería que provee capacidades multitasking para aplicaciones de control de hardware. FreeRTOS nos brinda un conjunto de documentos desarrollados y compilados en lenguaje C, donde algunos de los mismos son específicos para un puerto, mientras que otros son específicos para un solo puerto.

Bajo este contexto, con la finalidad de diseñar una rutina que permitiera la comunicación en tiempo real con la computadora Jetson Nano, se realizó la implementación de FreeRTOS para el diseño y ejecución de tareas, mismas que están estructuradas con el objetivo de brindar un entorno óptimo de ejecución de los comandos enviados por el Puzzlebot cuando los objetos de interés son identificados. Las tareas desarrolladas son las siguientes:

1. **receiveJetson.** Tarea definida con prioridad Realtime con la finalidad de ser la encargada de interpretar los datos de comunicación recibidos por la tarjeta STM32 desde la Jetson Nano. Su proceso de interpretación es basado en el diagrama 1, implementado por medio de un switch case que lee los dos bytes del buffer recibido de manera serial, de acuerdo con los cuales identifica la rutina a ejecutar y activa banderas para permitir que el resto de las tareas las interpreten e identifiquen si deben llevarse a cabo o no. Las banderas usadas son gripperFlag, artFlag y stopFlag. El código implementado se observa en la figura 6.

```
/* USER CODE END Header_StartReceiveJetson */
void StartReceiveJetson(void *argument)
{
  /* USER CODE BEGIN StartReceiveJetson */
  /* Infinite loop */
  for(;;)
  {
    read();
    switch (comm[0]){
        case 1:
            if (comm[1]==1)
              {
                gripperFlag = 1;
              }
            else if (comm[1]==2)
              {
                gripperFlag = 2;
              }
            else
              {
                gripperFlag = 0;
              }
            break;
        case 2:
            if (comm[1]==1)
              {
                artFlag = 1;
              }
            else if (comm[1]==2)
              {
                artFlag = 2;
              }
            else
              {
                artFlag = 0;
              }
            break;
        case 3:
            stopFlag = 1;
            break;
        default:
            gripperFlag = 0;
            artFlag = 0;
            stopFlag = 0;
            break;
        }
    osDelay(20000);
  }
  /* USER CODE END StartReceiveJetson */
}
```

Figure 6: Tarea receiveJetson.

2. **emStop.** Declarada con prioridad AboveNormal, pues es crucial que sea interpretada tan pronto como termina la ejecución de la tarea receiveJetson. El motivo de esta decisión se basa en el conocimiento de que, en caso de ser necesario que el sistema físico se detenga debido a la presencia de algún error complejo en el sistema principal, es importante que su rutina se ejecute previo a que alguna otra sea implementada. Esta tarea desactivará todas las banderas, de manera que el resto de las tareas identifiquen que sus rutinas no deben ser llevadas a cabo. Una vez que ha sido ejecutada activa la bandera stopOK. El código implementado se observa en la figura 7.

```
/* USER CODE END Header_StartEmStop */
void StartEmStop(void *argument)
{
  /* USER CODE BEGIN StartEmStop */
  /* Infinite loop */
  for(;;)
  {
      if(stopFlag == 1){
          gripperFlag = 0;
          artFlag = 0;
          stopFlag = 0;
      }
    osDelay(1);
  }
  /* USER CODE END StartEmStop */
}
```

Figure 7: Tarea emStop.

3. **artTask.** Esta tarea se define con una prioridad Normal1, ya que leerá las banderas previamente activadas y se encargará de posicionar adecuadamente la articulación de una manera específica en caso de que la tarea receiveJetson identifique esta orden enviada por la computadora Nano Jetson. La lectura se realiza con la bandera artFlag, donde los valores declarables son 1 para subir y 2 para bajar. Una vez que ha sido ejecutada activa la bandera artOK. El código implementado se observa en la figura 8.

```
/* USER CODE END Header_StartArtTask */
void StartArtTask(void *argument)
{
  /* USER CODE BEGIN StartArtTask */
  /* Infinite loop */
  for(;;)
  {
      if(artFlag == 1)
          {
              TIM2->CCR2 = artUp;
              osDelay(50);
              artFlag = 0;
              artOK = 1;
          }
      else if (artFlag == 2)
          {
              TIM2->CCR2 = artDown;
              osDelay(50);
              artFlag = 0;
              artOK = 1;
          }
      osDelay(1);
  }
  /* USER CODE END StartArtTask */
}
```

Figure 8: Tarea artTask.

4. **gripperTask.** Declarada con prioridad Normal2, con el objetivo de que su rutina sea ejecutada posterior a la rutina

de la articulación, en caso de que lo mismo sea indicado por la tarea receiveJetson. La bandera asociada en este caso es gripperFlag, donde los valores le indican si la solicitud recibida es abrir (1) o cerrar (2) las tenazas del gripper. Una vez que ha sido ejecutada activa la bandera gripperOK. El código implementado se observa en la figura 9.

```
/* USER CODE END Header_StartGripperTask */
void StartGripperTask(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    if(gripperFlag == 1)
    {
      TIM2->CCR1 = gripperOpen;
      osDelay(50);
      gripperFlag = 0;
      gripperOK = 1;
    }
    else if (gripperFlag == 2)
    {
      TIM2->CCR1 = gripperClose;
      osDelay(2000);
      gripperFlag = 0;
      gripperOK = 1;
    }
    osDelay(100);
  }
  /* USER CODE END 5 */
}
```

Figure 9: Tarea gripperTask.

5. **checkButton.** Tarea declarada con prioridad BelowNormal, su intención es leer de manera constante si entre las tenazas está sostenido un objeto, esto con la finalidad de que la Jetson Nano, una vez que ha recibido la información, decida el siguiente paso a ejecutar. Los valores brindados a la bandera objectHeld actúan de manera booleana, indicando si el objeto ha sido sostenido (1) o no (0). El código implementado se observa en la figura 10.

```
/* USER CODE END Header_StartCheckButton */
void StartCheckButton(void *argument)
{
  /* USER CODE BEGIN StartCheckButton */
  /* Infinite loop */
  for(;;)
  {
    if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_2) == GPIO_PIN_RESET){
        objectHeld = 0;
    }
    else{
        objectHeld = 1;
    }
    osDelay(1);
  }
  /* USER CODE END StartCheckButton */
}
```

Figure 10: Tarea checkButton.

6. **transmitJetson.** Última tarea en ejecutarse, por lo que se le asocia una prioridad Low. Es la encargada de realizar lecturas constantes de las banderas de confirmación (stopOK, artOK, gripperOK, objectHeld) y, con base en la interpretación de las mismas transmite el estado actual del gripper a la unidad Jetson Nano. Esta rutina almacenará los valores actuales de todas las banderas de confirmación en un arreglo de cuatro espacios y posteriormente las reiniciará todas. El código implementado se observa en la figura 11.

```
/* USER CODE END Header_StartTransmitJEtson */
void StartTransmitJEtson(void *argument)
{
  /* USER CODE BEGIN StartTransmitJEtson */
  /* Infinite loop */
  for(;;)
  {
    if (gripperOK == 1)
    {
      okFlags[0]= gripperOK;
      gripperOK = 0;
    }
    else
    {
      okFlags[0]= 0;
    }
    if (artOK == 1)
    {
      okFlags[1]= artOK;
      artOK = 0;
    }
    else
    {
      okFlags[1]= 0;
    }
    if (objectHeld == 1)
    {
      okFlags[2]= objectHeld;
    }
    else
    {-
      okFlags[2]= 0;
    }
    if (stopOK == 1)
    {
      okFlags[3]= stopOK;
      stopOK = 0;
    }
    else
    {
      okFlags[3]= 0;
    }
    osDelay(20100);
  }
  /* USER CODE END StartTransmitJEtson */
```

Figure 11: Tarea transmitJetson.

Todas estas tareas se comunican de manera piramidal por medio de las banderas declaradas y modificadas durante el proceso de ejecución del código. Esta comunicación es interpretada de manera visual en el diagrama 12.

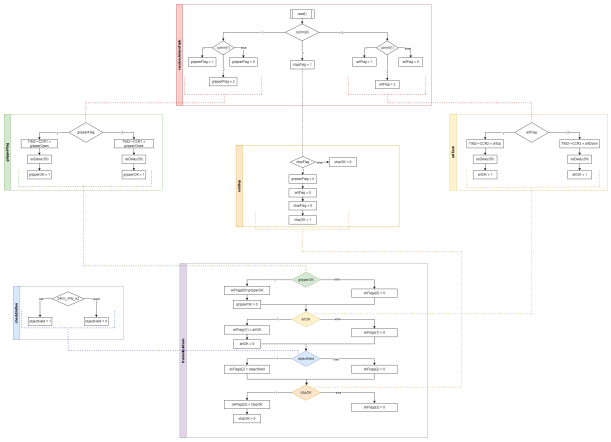

Figure 12: Esquema de comunicación entre tareas.

6. Comunicación entre Jetson Nano y STM32 a través de rosserial

La comunicación entre la computadora Jetson Nano y la tarjeta STM32F411CEU6 mediante la librería ROSserial desempeña un papel fundamental en el control autónomo del sis-

tema implementado en la STM32. Esta comunicación establece un enlace vital entre los dos componentes, permitiendo la transferencia de datos y comandos necesarios para una interacción eficiente y coordinada.

La Jetson Nano, como potente unidad de procesamiento, es capaz de realizar cálculos complejos y tomar decisiones inteligentes en tiempo real. Sin embargo, para que estas decisiones se traduzcan en acciones concretas en el mundo físico, es crucial la comunicación con la STM32, que actúa como interfaz de control para el sistema embebido. A través de la librería ROSserial, se establece una conexión bidireccional entre estos dos componentes, lo que permite el intercambio de datos y señales entre ellos de manera rápida y confiable.

La librería ROSserial brinda un marco de trabajo eficiente para la comunicación, facilitando la transferencia de mensajes y comandos entre la Jetson Nano y la STM32. Esto habilita el control autónomo del sistema implementado en la STM32, donde la Jetson Nano puede enviar instrucciones precisas y recibir información actualizada en tiempo real. De esta manera, se logra una colaboración efectiva entre ambos dispositivos, permitiendo la toma de decisiones inteligentes y la ejecución precisa de acciones en el sistema embebido.

RosSerialSTM32

Dentro de los paquetes que ofrece ROS, existe uno que posibilita la subscripción y publicación de tópicos a través de puertos seriales, llamado *rosserial*. Este paquete por sí solo contiene un nodo que recibe de parámetro un puerto específico donde se estará comunicando con el dispositivo a través de comunicación serial. Sin embargo, se necesita de otro paquete llamado *rosserial_stm32*, el cual contiene de ejemplos y códigos necesarios para generar las librerías y archivos de encabezado que se necesitan incluir en el proyecto de STM32 que será compilado. Las librerías son de propósito general, por lo que deben ser adecuadas para el modelo específico del microcontrolador. En este caso, se busca realizar una transmisión de datos entre una tarjeta de desarrollo STM32 y una Jetson Nano. La Jetson Nano tiene puertos USB capaces de realizar recibir datos por comunicación serial, sin embargo, la STM32F411CEU6 (el modelo específico utilizado), tiene varios puertos Rx y Tx programables para la transmisión asíncrona de datos. En este proyecto, se utilizaron los pines PB6 (Tx) y PB7 (Rx) para la comunicación USART.

Ya con los pines habilitados por la comunicación serial, fue necesario agregar los archivos en las carpetas *Inc* y *Src* del proyecto, creados con el paquete de ROS mencionado previamente. Todos los archivos generados con el paquete de ROS, se encuentran en C++, por lo que si el proyecto se encuentra programado en lenguaje C, habrá que realizar algunas modificaciones de compatibilidad. Una vez modificados los archivos para que sean compatibles con el proyecto, se puede utilizar la notación de un nodo de C++ en ROS para comunicarse entre tópicos por medio del puerto serial. El código implementado en la tarjeta utilizada para el desarrollo del proyecto es el mostrado en la Figura 14.

En esa sección de código, se declaran las variables necesarias para el correcto funcionamiento de un nodo de ROS y pueda

```
static void MX_USART1_UART_Init(void)
{

  /* USER CODE BEGIN USART1_Init 0 */

  /* USER CODE END USART1_Init 0 */

  /* USER CODE BEGIN USART1_Init 1 */

  /* USER CODE END USART1_Init 1 */
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 57600;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART1_Init 2 */

  /* USER CODE END USART1_Init 2 */

}
```

Figure 13: Configuración de la comunicación serial.

```
ros::NodeHandle nh;

void req(const std_msgs::String& msg);

std_msgs::String str_msg;
ros::Publisher chatter("chatter", &str_msg);
ros::Subscriber<std_msgs::String> stm32_comms("gripper_action", &req);
std::string request="";
std::string gripState="";

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart){
  nh.getHardware()->flush();
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
  nh.getHardware()->reset_rbuf();
}
/* USER CODE END PV */
```

Figure 14: Creación del nodo de ROS en C++.

ejecutar acciones de publicación y subscripción a tópicos. En este caso, se declara un publicador llamado *chatter* que publica mensajes de tipo String en un tópico del mismo nombre. El tópico *chatter* es donde la STM32 va a actualizar el último estado del brazo robótico, es decir, subir el brazo, bajar el brazo, cerrar el agarre o abrir el agarre. Por otra parte, también se declara el tópico *gripper_action*, en el que la Jetson Nano estará publicando las acciones que tiene que llevar a cabo el gripper, las cuales son las mismas 4 acciones descritas anteriormente. Esta comunicación es posible gracias al nodo *serial_node*, el cual se encarga de leer los datos recibidos por el puerto serial y publicar esos datos en sus respectivos tópicos. El diagrama de comunicación entre los nodos de la Jetson Nano y la STM32 se encuentra en la Figura 15.

Dentro de la programación de la STM32, existen tres funciones que son fundamentales para el funcionamiento correcto de la comunicación entre nodos. La primera es la función *setup()*, que se encarga de inicializar el nodo *nh* y de añadir los subscriptores y publicadores, es decir, de establecer la comunicación con los tópicos *chatter* y *gripper_action*. En segunda, se encuentra la función que se encarga de publicar el estado del gripper en el tópico *chatter*, la cual está catalogada como *jetsonResponse()*. Por último, se encuentra la función *req()* que está constantemente obteniendo cualquier actualización del tópico *gripper_action*, y en dado caso, guardar la acción solicitada por la Jetson Nano en su respectiva variable. Las funciones descritas se pueden encontrar en la Figura 16.

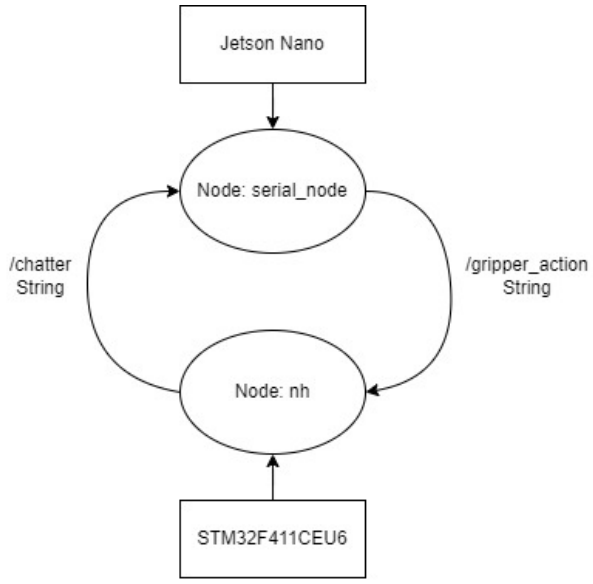

Figure 15: Diagrama de comunicación con tópicos.

```
void setup(void)
{
  nh.initNode();
  nh.advertise(chatter);
  nh.subscribe(stm32_comms);
}


void jetsonResponse(void)
{
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    const char* str = gripState.c_str();
    str_msg.data = str;
    chatter.publish(&str_msg);
    nh.spinOnce();
}

void req(const std_msgs::String& msg){
    request = msg.data;
}
```

Figure 16: Funciones del nodo nh.

Finalmente, existe la tarea *StartReceiveJetson* que se encarga de modificar las banderas de las otras tareas con base en los mensajes recibidos del tópico *gripper_action*. Por cuestiones de practicidad para las pruebas, en esta ocasión se utilizaron caracteres clave para identificar el tipo de acción que solicitaba la Jetson Nano a través del tópico. Las 5 instrucciones que se determinaron con sus respectivos caracteres clave son: levantar el brazo, con el comando ”au”; bajar el brazo, con el comando ”ad”; cerrar el agarre, con el comando ”gc”; abrir el agarre, con el comando ”go”; y finalmente, detener el gripper, con el comando ”s”. La programación para la modificación de las banderas con base en el comando recibo por medio del tópico *gripper_action* se puede visualizar en la Figura 17.

En la tarea *StartReceiveJetson* solo se cambian los valores de las banderas con las que otras tareas ya se encargan de hacer sus respectivos procesos. Es por eso, que la función *jetsonResponse()* se llama cada vez que se ejecuta el comando obtenido a través del tópico *gripper_action* en su respectiva tarea. Además, previo a su llamada, también se modifica el valor de la variable *gripState* que se publica en el tópico *chatter* donde el nodo de la STM32 publica el estado actual del gripper. Por ejemplo, en la Figura 50 se puede observar cómo es llamada la función

```
void StartReceiveJetson(void *argument)
{
  /* USER CODE BEGIN StartReceiveJetson */
  /* Infinite loop */
  for(;;)
  {
        switch (request[0]){
            case 'g':
                if (request[1] == 'o')
                    {
                        gripperFlag = 1;
                    }
                else if (request[1] == 'c')
                    {
                        gripperFlag = 2;
                    }
                else
                    {
                        gripperFlag = 0;
                    }
                break;
            case 'a':
                if (request[1] == 'u')
                    {
                        artFlag = 1;
                    }
                else if (request[1] == 'd')
                    {
                        artFlag = 2;
                    }
                else
                    {
                        artFlag = 0;
                    }
                break;
            case 's':
                    stopFlag = 1;
                    break;
            default:
                    gripperFlag = 0;
                    artFlag = 0;
                    stopFlag = 0;
                    break;
          }
        osDelay(200);
  }
  /* USER CODE END StartReceiveJetson */
}
```

Figure 17: Código de la tarea StartReceiveJetson.

jetsonResponse() cada vez que se realiza el movimiento hacia arriba o abajo del brazo robótico.

```
void StartArtTask(void *argument)
{
  /* USER CODE BEGIN StartArtTask */
  /* Infinite loop */
  for(;;)
  {
      if(artFlag == 1)
          {
              TIM2->CCR2 = artUp;
              osDelay(50);
              artFlag = 0;
              artOK = 1;
              gripState = "artUp";
              jetsonResponse();
          }
      else if (artFlag == 2)
          {
              TIM2->CCR2 = artDown;
              osDelay(50);
              artFlag = 0;
              artOK = 1;
              gripState = "artDown";
              jetsonResponse();
          }

      osDelay(100);
  }
  /* USER CODE END StartArtTask */
}
```

Figure 18: Ejemplo de tarea con llamada a la función *jetsonResponse()*.

7. Conclusiones

Como pudimos observar, la utilización de sistemas embebidos en aplicaciones robóticas automatizadas ofrece numerosas ventajas. En primer lugar, brinda la capacidad de procesamiento local, lo que permite realizar cálculos y toma de decisiones en tiempo real en el propio robot. Esto resulta esencial en entornos dinámicos donde la latencia y la velocidad de respuesta son críticas. Además, la implementación de sistemas embebidos proporciona un nivel adicional de seguridad y confiabilidad, ya que el control y las operaciones se realizan internamente en el robot, sin depender de conexiones externas o servicios en la nube.

Aunado a ello, la integración de ROSserial en sistemas embebidos permite aprovechar las capacidades de ROS, como la planificación de trayectorias, el mapeo, la navegación y la percepción, para mejorar la autonomía y la inteligencia de los robots. La capacidad de comunicarse con otros nodos ROS a través de ROSserial facilita la interacción con sistemas externos, como sensores, actuadores y sistemas de control. Esto promueve la colaboración y la integración de diferentes componentes y tecnologías, permitiendo la creación de sistemas robóticos más sofisticados y completos.

Durante el desarrollo del proyecto, se tuvieron complicaciones de diferentes tipos; de software, con la compatibilidad de las librerías utilizadas o en la misma programación del brazo robótico; o físicas, debido a que el diseño no tenía la rigidez requerida para el proyecto y fue necesario realizar reajustes y sustitución de partes. Además, hay que agregar a esto que se tuvieron que programar el microcontrolador con FreeRTOS, un sistema operativo con el que no se tiene mucha familiaridad, y cuya implementación no es muy sencilla debido a la complejidad para visualizar el comportamiento del sistema en funcionamiento. A pesar de estas y más complicaciones, fue posible generar un modelo de brazo robótico completamente funcional para la tarea para la que fue programado. El brazo robótico fue capaz de soportar el peso de los cubos y transportarlos con éxito, además de comunicarse con una Jetson Nano para poder recibir comandos y generar una respuesta a esos comandos por medio de tópicos de ROS. En conclusión, a pesar de que existan áreas de mejora en cuanto al diseño y programación del brazo robótico, se alcanzó con éxito el objetivo establecido y cumplió correctamente su cometido.

De igual manera, en la figura 19 se muestra el diseño digital del gripper empleado para alcanzar los objetivos propuestos por el módulo.

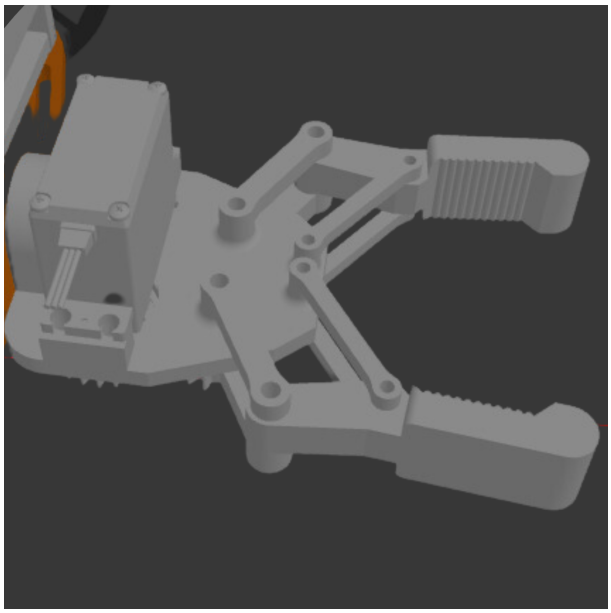

Figure 19: Diseño de gripper.

8. ANEXO: Pruebas unitarias

Realizar pruebas unitarias es de vital importancia en el desarrollo de software. Estas pruebas se centran en verificar el funcionamiento correcto de las unidades individuales de código, como funciones, métodos o clases, de forma aislada. La importancia de las pruebas unitarias radica en varios aspectos clave. En primer lugar, garantizan que cada unidad de código cumpla con los requisitos y las especificaciones establecidas, lo que ayuda a prevenir errores y asegurar la calidad del software. Además, las pruebas unitarias facilitan la detección temprana de posibles defectos, ya que permiten identificar y corregir errores en etapas tempranas del desarrollo, lo que reduce los costos y el tiempo requerido para su solución. Asimismo, las pruebas unitarias brindan un nivel de confianza en el código, ya que proporcionan evidencia objetiva de su correcto funcionamiento. Esto es especialmente relevante cuando se realizan cambios o modificaciones en el código, ya que las pruebas unitarias permiten verificar que no se introduzcan errores inadvertidos. Otro beneficio importante de las pruebas unitarias es que proporcionan una base sólida de evidencia objetiva sobre el funcionamiento del código. Al contar con un conjunto exhaustivo de pruebas que validan cada unidad de código, se puede tener confianza en que el software cumple con los estándares de calidad establecidos. A continuación se muestran las fichas técnicas de todas las pruebas unitarias realizadas, incluyendo su respectivo enlace a espacio de github y video demostrativo.

References

Barry, R. (2009). FreeRTOS reference manual: API functions and configuration options. Real Time Engineers Limited.

Chen-Kai Lin, Bow-Yaw Wang. (2022). Analyzing FreeRTOS Scheduling Behaviors with the Spin Model Checker.

Bouyssounouse, Bruno and Sifakis, Joseph (2005). Embedded Systems Design: The ARTIST Roadmap for Research and Development. Springer-Verlag.