Sistema de conducción inteligente para tractores agrícolas

Tovar Hectr A00840308, Llamas Paola A01178479, Domínguez José A01285873

TE2003B- Diseño de sistemas en chip
Enrique González Guerrero
Raúl Peña Ortega
Gpo 601

Palabras clave - RPM, John Deere, velocidad angular, tractor, radio, relación de transmisión, python, simulación computacional y sistema embebido.

I. Resumen

La industria agrícola enfrenta diversos retos a los cuales se deben a la escasez de mano de obra y la necesidad de aumentar la eficiencia en las operaciones de cultivo. Una empresa líder como viene siendo John Deere ha desarrollado tractores autónomos que utilizan sistemas embebidos avanzados para la automatización de tareas agrícolas que tiene como principal objetivo combatir los retos que se enfrenta y consiguiendo mejorar la precisión y reducir la dependencia de operadores humanos. Estos tractores han demostrado ser efectivos en la optimización de procesos como la siembra y la cosecha. Aumentando la productividad y sostenibilidad de las explotaciones agrícolas.

La solución propuesta en este proyecto es el desarrollo de un sistema de simulación que modela el comportamiento de un tractor mediante la generación aleatoria de datos relacionados con la velocidad angular, el radio de la rueda y la relación de transmisión. Usando estos parámetros se calculará el valor de las revoluciones por minuto conocidas como RPM.

Este modelo se implementa en Python, utilizando bibliotecas como random, pandas, math y matplotlib para automatizar la generación de datos y luego poder almacenarlos en formato .csv, y representar gráficamente el comportamiento del tractor en tiempo real. Los resultados obtenidos muestran cómo pequeñas variaciones en la velocidad angular y el radio de la rueda logran afectar significativamente el valor de RPM. A través de varias simulaciones se observó un comportamiento coherente con el modelo físico esperado.

II. Introducción

2.1.Contexto general

John Deere es una empresa líder en la fabricación de maquinaria agrícola, reconocida por su innovación constante y su papel en la transformación digital del campo. Esto es gracias al desarrollo de tractores autónomos ya que son integradas con tecnologías como vienen siendo los sistemas embebidos y la visión computacional ya que ha ayudado a aumentar la eficiencia y a reducir la dependencia de operadores humanos [1].

Pero lo importante es sobre el uso de los sistemas embebidos. Los sistemas embebidos se utilizan mayormente en la industria automotriz ya que tienen el objetivo de controlar funciones como el frenado, la dirección y la gestión del motor. En este caso que nos enfocamos en los tractores, específicamente los de John Deere permiten automatizar procesos complejos como el guiado preciso, la gestión de combustible y la optimización de rutas de cultivo. Para concluir los sistemas embebidos se colocan dentro del sistema más grande para realizar tareas o funciones especificadas [2].

2.2.Delimitación del objeto de estudio

Este trabajo tiene como meta lograr centrarnos específicamente en el diseño y en una simulación de un sistema embebido que logre modelar la conducción de un tractor autónomo. Ya que fue desarrollado en un entorno simulado

mediante Python y los resultados del cálculo de revoluciones por minuto (RPM) en distintas condiciones fueron analizados meticulosamente para asegurar tener datos coherentes y válidos. No se utilizarán datos reales del entorno físico y las condiciones meteorológicas o del terreno no están siendo consideradas lo cual aplicarlo en un entorno realista puede ser una limitante, junto con la cuestión de que se trabaja a contrarreloj.

2.3.Planteamiento del problema

Como sociedad y planeta nos estamos enfrentando a varios retos y problemas siendo uno de ellos la escasez de operadores calificados y el crecimiento de la demanda agrícola, como se logra apreciar en la figura 1.

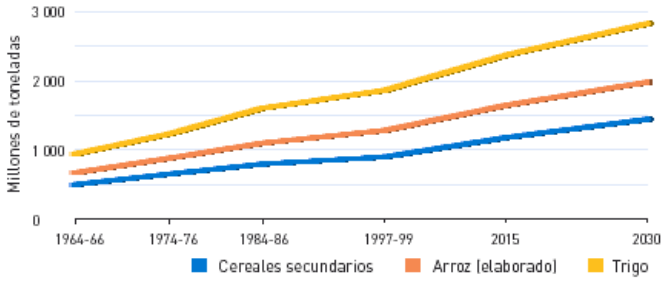


Figura 1. Demanda agrícola, (Agricultura Mundial, 2015)

Debido a esos problemas se ha creado la necesidad de automatizar procesos con sistemas inteligentes. Estos sistemas deben ser capaces de manejar tareas complejas en los entornos a los que están expuestos y es aquí donde los sistemas embebidos permiten modelar la conducción de tractores. Los sistemas embebidos logran mejoras en precisión, rendimiento y seguridad. Ayudando a modelar y a eficientar los tractores aprovechando su rendimiento al máximo.

2.4.Justificación

El uso de sistemas embebidos nos sirve para facilitar la innovación en este caso en los tractores inteligentes de John Deere. Gracias a su bajo consumo energético, tamaño compacto y capacidad de procesamiento en tiempo real los sistemas embebidos son la solución ideal a este problema. Convirtiéndolos en una base ideal para el desarrollo de soluciones autónomas en la agricultura.

Una de las áreas clave de la nueva era de la tecnología agrícola es la agricultura de precisión. Esta se caracteriza por el uso de tecnología para monitorear, analizar y tomar medidas basadas en diversos parámetros del cultivo, con alta precisión. Los tractores autónomos aprovechan la visión artificial, la telemática, algoritmos de aprendizaje profundo y aplicaciones móviles en la nube para lograrlo [4].

2.5.Marco teórico

2.5.1. Conceptos relevantes

Un sistema en chip (SOM) es un circuito integrado que comprime todos los componentes necesarios del sistema en una sola pieza de silicio. Al eliminar la necesidad de componentes de sistema grandes y separados, los SoC simplifican el diseño de la placa de circuito, lo que resulta en mayor potencia y velocidad sin comprometer la funcionalidad del sistema [5].

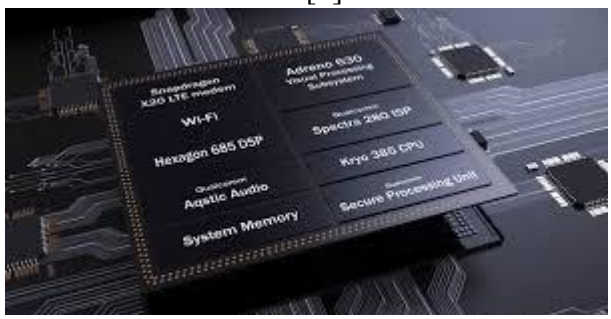


Figura 2. Sistema en chip, (¿Qué Es Un System on a Chip?, 2020)

"Un microcontrolador es un circuito integrado pequeño que contiene una unidad central de procesamiento (CPU), memoria y periféricos de entrada/salida, todo en un solo chip, diseñado para controlar tareas específicas dentro de un sistema embebido Estos circuitos integrados compactos (IC) contienen un núcleo de procesador (o núcleos), memoria de acceso aleatorio (RAM) y memoria de solo lectura programable borrable eléctricamente (EEPROM) para almacenar los programas personalizados que se ejecutan en el microcontrolador, incluso cuando la unidad está desconectada de una fuente de alimentación"[6].

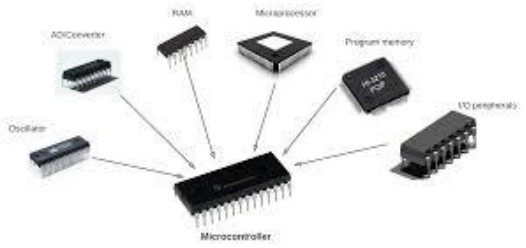


Figura 3. Microcontrolador,(Ariat-Tech.com, 2024)

Los microcontroladores y los sistemas en chip nos permiten crear productos más rápidos, compactos y económicos siendo de gran utilidad para la creación de nuevos productos. Los sistemas embebidos se encuentran en automóviles, electrodomésticos, robots, dispositivos médicos y más. Entre sus ventajas están el bajo consumo energético y la rapidez de respuesta. Sin embargo, presentan desafíos como capacidad limitada de procesamiento y dificultad para actualizar el software siendo una de sus desventajas más relevantes [8].

Un RTOS significa sistema operativo en tiempo real que es el encargado de gestionar tareas que requieren respuestas rápidas y garantizadas como las que se necesitan en conducción autónoma. Permite ejecutar múltiples procesos en paralelo, respondiendo a eventos en tiempo real. El objetivo principal de un RTOS es ejecutar operaciones críticas de forma oportuna. Garantiza que ciertos procesos se completen dentro de plazos estrictos, lo que lo hace ideal para aplicaciones donde la sincronización es crucial. También es útil para la multitarea y el trabajo basado en tareas [9].

Un RTOS ofrece ventajas importantes como la capacidad de dividir aplicaciones en partes más manejables, reducir el tiempo total de desarrollo y proporcionar funciones listas para usar, como manejo de tareas, prioridades, sincronización y comunicación entre tareas. Además que permite implementar multitarea y control de prioridades, lo que resulta ideal para aplicaciones que requieren respuestas en tiempo real. Sin embargo también presenta desventajas no es tan potente como un sistema operativo completo, no proporciona controladores de bajo nivel, limita el acceso directo del desarrollador a la aplicación, carece de compatibilidad con algunos periféricos de microcontroladores y no se recomienda para sistemas que requieren un alto nivel de multitarea. Por el otro lado el diseño bare metal, aunque más simple y rápido para tareas únicas presenta dificultades para escalar y mantener aplicaciones complejas, lo que lo hace menos flexible frente a sistemas embebidos que requieren múltiples procesos simultáneos [10].

“Linux es el nombre que recibe el núcleo (kernel) y en general una serie de sistemas operativos de tipo Unix bajo la licencia GNU GPL (General Public License o Licencia Pública General de GNU) y similares. En muchos aspectos, Linux es similar a otros sistemas operativos como Windows, macOS (antes OS X) o iOS. Al igual que ellos, Linux puede tener una interfaz gráfica y los mismos tipos de software de escritorio a los que estás acostumbrado, como procesadores de texto, editores de fotos, editores de vídeo, etc. Pero Linux también se diferencia en muchos aspectos importantes. En primer lugar, y quizás su característica más importante, es un software de código abierto. El código utilizado para crear Linux es gratuito y está disponible para que el público lo vea, lo edite y -para los usuarios con los conocimientos adecuados- contribuya a él” [11].

Linux embebido se caracteriza por ser de código abierto, escalable y con bajo consumo de energía, lo que lo hace ideal para dispositivos alimentados por batería. Entre sus principales ventajas se encuentran la posibilidad de desarrollar soluciones personalizadas y económicas, su capacidad de adaptación a distintas arquitecturas de hardware, la personalización avanzada de aplicaciones y el respaldo de una gran comunidad de usuarios y desarrolladores. No obstante, también presenta ciertas desventajas como los tiempos de arranque ya que pueden ser prolongados, lo que representa un inconveniente en aplicaciones que requieren inicio inmediato su estructura compleja puede dificultar el desarrollo requiere una cantidad considerable de memoria, lo cual puede limitar su uso en hardware muy restringido y el hecho de ser software de código abierto puede plantear retos al momento de obtener certificaciones para aplicaciones críticas. Linux embebido sigue siendo una de las opciones más utilizadas en sectores como la automatización industrial, el control de maquinaria, el software de vuelo, la navegación satelital, entre otras. [12]

La programación en sistemas Linux embebidos suele apoyarse en lenguajes como Bash y Python debido a su eficiencia y versatilidad. Los scripts de Bash son fundamentales para automatizar tareas del sistema, como la inicialización de servicios, la gestión de procesos y la configuración

de hardware al arranque. Su sintaxis simple nos permite poder crear secuencias de comandos que controlan directamente el sistema operativo pero es útil cuando no se usa para cosas tan complejas. Por otro lado está Python que es ampliamente utilizado en sistemas embebidos por su legibilidad y su amplia variedad de bibliotecas ya que nos ayudan a facilitar el manejo de sensores hasta el procesamiento de datos y la creación de interfaces gráficas [13].

2.5.2. Revisión de literatura

La integración de sistemas embebidos en la agricultura ha sido objeto de varios estudios ya que es un tema que constantemente se busca una mejor y tener una evolución. Con el objetivo de transformar las prácticas agrícolas tradicionales. Un estudio reciente presentó un sistema embebido innovador para el mapeo del rendimiento de implementos en tractores, utilizando microcontroladores y sensores para digitalizar y registrar parámetros de rendimiento en tiempo real [14]. Esta solución nos ofrece poder tener una alternativa rentable para monitorear y mejorar la eficiencia operativa de la maquinaria agrícola.

Un avance bastante significativo ha sido la incorporación de los sistemas embebidos ya que ha sido fundamental en la revolución de la inteligencia artificial aplicada a tractores autónomos. Los sistemas de cámaras embebidas permiten a estos tractores capturar datos visuales esenciales, facilitando la toma de decisiones ya que cuentan con información suficiente por parte de los agricultores [15]. Esta tecnología mejora la precisión en operaciones como la siembra y la cosecha y ayuda a la optimización del uso de recursos y reduciendo desperdicios.

Estos estudios son un parámetro para nosotros y nos ayudan a ver el rol que están teniendo los sistemas embebidos en la evolución hacia una agricultura más automatizada y eficiente.

2.6.Objetivos

Nuestro objetivo general es poder desarrollar un sistema de simulación basado en sistemas embebidos que logre modelar la conducción de un tractor autónomo y que la generación de datos se logre almacenar y graficar para una mejor visualización.

Como objetivos específicos son los siguientes:

- Desarrollar un programa en Python que genere datos aleatorios para las variables de velocidad angular de la rueda, radio de la rueda y relación de transmisión. Que logre calcular las RPM correspondientes.
- Poder implementar un modelo matemático que sea capaz de calcular las RPM del tractor.
- Nuestro último objetivo específico sería crear una interfaz gráfica que nos permita visualizar en tiempo real las gráficas del comportamiento del tractor con el objetivo de facilitar el análisis de los datos generados por nuestro modelo matemático.

2.7.Hipótesis

Nuestra hipótesis consiste en que el uso de sistemas embebidos facilitará la simulación y proyección del comportamiento dinámico de un tractor, permitiendo analizar con mayor precisión variables clave como las revoluciones por minuto (RPM), la velocidad angular y el radio de la rueda en distintas condiciones operativas.

III. PROPUESTA

3.1.Metodología

El desarrollo del sistema se abordó con un enfoque iterativo, propio del diseño de sistemas embebidos. Esto implicó avanzar paso a paso, asegurando que cada módulo funcionara correctamente antes de realizar la integración total. Este método nos ayudó a entender mejor la funcionalidad de cada componente, facilitando la localización de errores y su corrección de forma oportuna.

El primer paso consistió en establecer los roles de cada componente del sistema iniciando com el microcontrolador STM32 se encargó de leer los sensores de aceleración que viene siendo el potenciómetro y freno que es el botón, controlar la señal PWM hacia los LEDs y actualizar el contenido del display LCD. La

ESP32 fue configurada para recibir los datos mediante comunicación serial UART, ejecutar el modelo matemático de transmisión automática, y enviar los resultados usando el protocolo MQTT. Finalmente la Raspberry Pi recibió la información para almacenarla en un archivo CSV y graficarla en tiempo real.

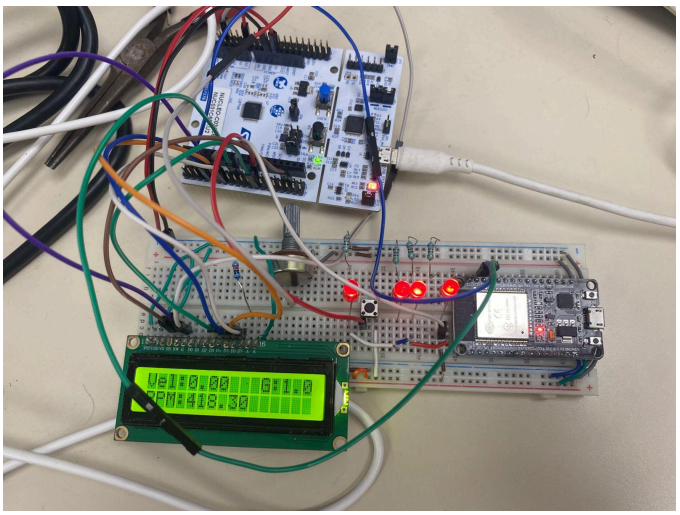


Figura 4. Primer prototipo

Antes de la implementación de código, se realizó el armado del circuito en protoboard como se ve en la Figura 4, verificando las conexiones físicas según el diagrama esquemático. Se realizaron pruebas individuales con cada sensor, utilizando un convertidor FTDI para observar las transmisiones de datos. En la STM32 el código fue desarrollado en C, iniciando con la lectura de entradas analógicas y digitales, y continuando con la generación de señales PWM.

Cada integrante del equipo asumió responsabilidades en distintas etapas del sistema. Una vez validada la lectura de datos, se avanzó a la programación del envío por UART. Posteriormente, se implementó en la ESP32 la recepción de estos datos y su reenvío vía Wi-Fi. En paralelo, se trabajó en un script de Python en la Raspberry Pi para almacenar los datos y graficarlos utilizando librerías como matplotlib.

El desarrollo se organizó en ciclos cortos. En cada ciclo se probaban funciones individuales, como la lectura del potenciómetro o el control de intensidad de los LEDs, antes de integrarlas. Posteriormente, se añadieron los módulos de comunicación que el UART y MQTT, lo cual representó uno de los principales retos del proyecto. Cada avance se validó mediante pruebas funcionales, lo que evitó errores acumulados y permitió mantener una integración estable a lo largo del proyecto.

La fase de conceptualización comenzó con la idea de simular una transmisión automática en un entorno embebido. Luego se desarrollaron los prototipos individuales que era la lectura de sensores, control de salidas como los LEDs y la LCD, después estaba la comunicación UART y visualización en Python. Una vez validados por separado, los módulos fueron integrados en un sistema completo logrando que la ESP32 actuará como puente entre el STM32 y la Raspberry Pi. Esta integración permitió una simulación fluida del sistema con comunicación en tiempo real y representación visual de datos relevantes como aceleración, velocidad y marcha.

Finalmente, en la etapa final del prototipo se empezó con el diseño del chasis por lo que utilizamos un chasis, que se muestra a continuación:

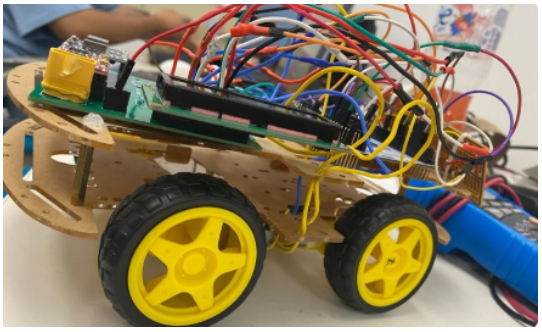


Figura 5. Chasis montado

Siguiendo con el desarrollo del prototipo se realizó una placa PCB para evitar la cantidad de conexiones, en donde en esta PCB se consideraba lo siguiente:

- 2 Drivers TB6612FNG
- 2 Reguladores MINI560
  - 1 Configuración de 12Vout
  - 1 Configuración de 5Vout
- NodeMCU
- LCD de 16 pines
- 4 Motores amarillos
- 2 potenciometros

- 1 Entrada XT60 para la fuente de alimentación
- 1 resistencia de 220Ω
- 4 Borneras-terminales para los motores
- 5 JST de 4 pines
  - INA de los 4 motores
  - INB de los 4 motores
  - PWM de los 4 motores
  - LCD2
- 2 JST de 3 pines
  - RX
  - TX
  - Potenciometro
  - LCD1
- 1 JST de 2 pines
  - Vin para la STM32

Así mismo se ideaba el usar una PCB embebida en donde estaría el NodeMCU, la cual ya estaba planeada en la placa, y la STM32 pero únicamente se puede embeber el procesador por lo que se descarto la idea por su complejidad.

Al final por cambio de microcontrolador y falta donde inicialmente la placa consideraba un NodeMCU pero mientras se desarrollaba el proyecto la ESP-32 terminó siendo el microcontrolador para el proyecto por lo que la ranura en la PCB del NodeMCU se dejó de utilizar, así mismo hubo falta de espacio para el potenciómetro .

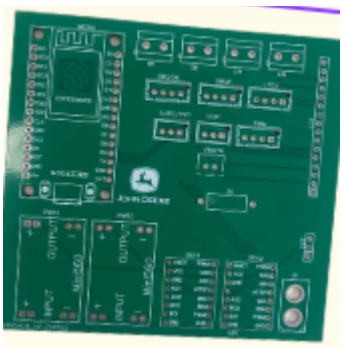


Figura 6. Placa PCB

Por consiguiente añadimos una placa de cobre donde integramos los componentes faltantes, la ESP-32, los dos potenciómetros y el botón.

Finalmente nuestro prototipo después de varias iteraciones se logró un prototipo sólido y eficiente.

Figura 7. Prototipo Final

3.2.1 Uso de lenguajes de programación

Para este entregable se utilizaron dos lenguajes de programación principales los cuales son C embebido, C++ y Python.

El lenguaje C embebido fue utilizado para la programación de bajo nivel del microcontrolador STM32. A través de este lenguaje se implementó el código necesario para la lectura del potenciómetro y del push button que vendría siendo la aceleración y freno. Aparte de eso está el envío de datos por UART, la generación de señales PWM para el control de LEDs, así como la interfaz mediante un display LCD. El uso de C nos permite poder programarlo [16].

El lenguaje de Python fue empleado en el uso de la Raspberry Pi para cuando reciba los datos transmitidos vía protocolo MQTT, y los almacena en un archivo CSV para después generar los gráficos en tiempo real. Se utilizaron bibliotecas como matplotlib para la visualización de la información.

C++ fue usado para poder programar la ESP32 y así poder recibir los datos de la STM32 y también para enviar los datos a la Raspberry Pi usando el protocolo MQTT.

Gracias a la combinación de estos lenguajes, nos permitió crear un sistema embebido distribuido, con procesamiento en tiempo real en el microcontrolador y visualizar los datos en la Raspberry Pi.

3.2.2 Recursos utilizados del STM32 y Raspberry Pi

Para la comunicación serie se habilitó USART entre el STM32 y el ESP32. Se configuró a 115200 baud formato 8 bits de datos, sin paridad y 1 bit de parada. Con el bus principal a 64 MHz, el registro BRR quedó en 0x022B y en CR1 se activaron transmisión,

recepción y la interrupción TXE para vaciar el búfer sin bloquear la CPU. Los pines utilizados fueron PA9 (TX) y PA10 (RX).

Para el control de los cuatro motores se empleó TIM1 en modo PWM (canales CH1–CH4). Donde iba de 0-100 % la potencia de los motores. El bit MOE en BDTR habilitó la salida, y cada canal alimenta los pines PB4, PB5, PB0 y PB1. El ciclo útil se actualiza en la tarea de control (T2).

La adquisición analógica se realiza con ADC1 a 12 bits sobre el canal 0 (PA0), donde está el potenciómetro de aceleración. El tiempo de muestreo se estableció en 71,5 ciclos para atenuar el ruido. Un evento update del propio TIM1 dispara la conversión al inicio de cada frame de 1 kHz garantizando coherencia entre la lectura y la salida PWM. El SysTick genera el pulso base del sistema cada 1 ms: LOAD = 64 000-1.

Como interfaz local, se conectó un LCD 16×2 en modo de 4 bits. Las líneas RS, RW, E y D4-D7 se asignaron a PB9-PB14 la pantalla muestra RPM y velocidad.

La Raspberry Pi 4 actúa como broker Mosquitto en el puerto. Recibe por Wi-Fi (2,4 GHz) los mensajes MQTT publicados por el ESP32. Genera un script Python con la librería y almacena los datos en CSV y genera gráficas en tiempo real con matplotlib, brindando monitorización remota sin interferir.

LCD	**STM32**
RS	PB9
RW	PB10
E	PB11
D4	PB12.2
D5	PB13
D6	PB14
Potenciometro	PA0
LEDs	PB4
Botón	PA7

Tabla 1. Conexiones

3.2.3 Protocolos de Comunicación

Para este trabajo los protocolos de comunicación que fueron implementados fueron el UART y el MQTT. Cada uno seleccionado en función del tipo de interacción entre los dispositivos involucrados y sus requerimientos.

Iniciando con el UART que viene de Universal Asynchronous Receiver-Transmitter [17], se empleó como protocolo de comunicación serial entre el microcontrolador que viene siendo la STM32 y la ESP32. Esta interfaz nos permite la transmisión de datos en formato de texto, facilitando el envío de información recolectada por el potenciómetro y el botón que vendría siendo la aceleración y el freno, desde la STM32 hacia la ESP32, así como la recepción de datos desde la ESP32 hacia la STM32.

MQTT que viene siendo Message Queuing Telemetry Transport que es un protocolo de mensajería ligero basado en el modelo publicador-suscriptor [18]. El protocolo de comunicación de MQTT es ideal para dispositivos con recursos limitados y redes inestables. En este proyecto se utilizó para transmitir de forma inalámbrica la información recibida por la ESP32 hacia la Raspberry Pi, donde se almacenan los datos y se visualizan en tiempo real.

Estos protocolos permiten construir una arquitectura más eficiente haciendo que puedan interactuar en tiempo real para poder simular el funcionamiento de una transmisión automática.

3.2.4 Diagramas, esquemáticos y conexiones del prototipo

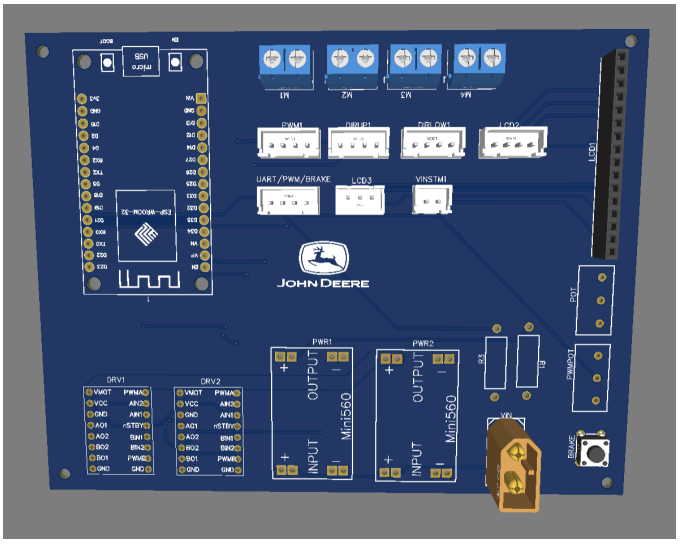


Figura 8. PCB Optimizada e Ideal

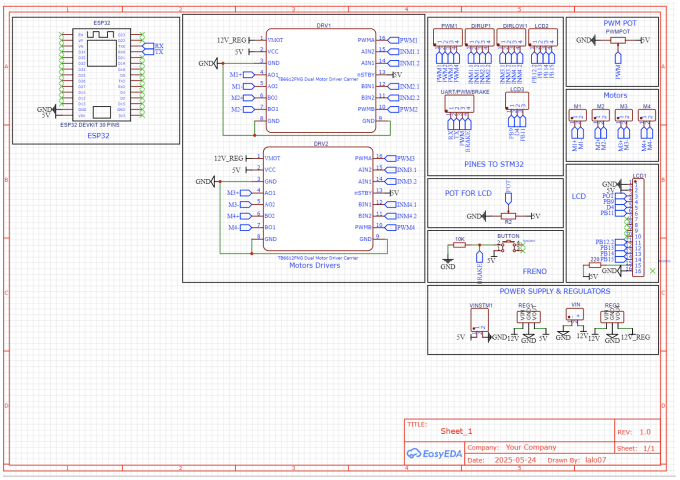


Figura 9. Diagrama esquemático

Como pueden ver en la Figura 8, es una placa diferente y diseño al presentado en el apartado 3.1, esto debido a que este es el diseño óptimo para nuestro prototipo en donde se considera una arquitectura modular construida para una ESP32, este microcontrolador fue seleccionado por sus características que nos ayudan en nuestro sistema, principalmente por su WiFi. En el esquema, se observa una clara distribución de bloques funcionales que permiten integrar los componentes los cuales son los ESP-32, STM32, LCD, Potenciómetros y Motores.

La alimentación del sistema se basa a través de una entrada de 12V, que se distribuye en dos, logico y motores, es decir, 12V y 5V, en donde lógico se usa para los componentes y los microcontroladores y motores recibe directamente 12V aunque se regula para evitar picos de amperaje o de voltaje.

Se implementa un potenciómetro conectado a una entrada analógica del microcontrolador. Este componente permite dar una señal variable que será utilizada para controlar los motores.

Asimismo, el sistema cuenta con una pantalla LCD. Esta pantalla permite al usuario visualizar información relevante del sistema en tiempo real como la aceleración, velocidad del motor y la marcha. También está el botón que es el freno. Para mejor visualización revisar el apartado de anexos.

3.2.5 Diagramas de flujo de programación del STM3

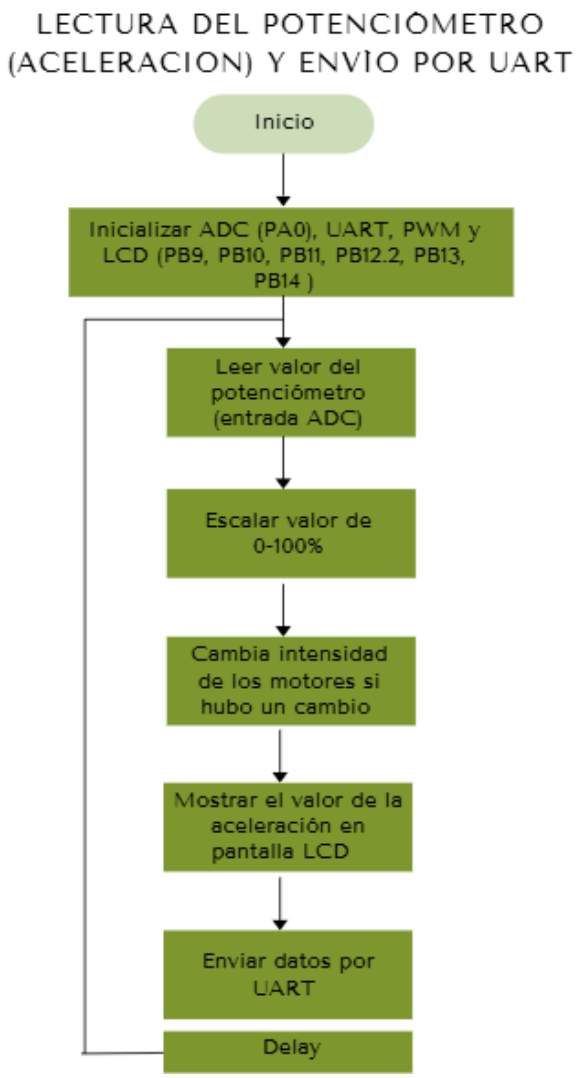


Figura 10. Lectura del Potenciómetro

En el arranque se configuran los periféricos con llamadas como ADC_Init(PA0), UART_Init(USART1,115200), PWM_Init(TIM1,1kHz) y LCD_Init(). A continuación entra en un bucle infinito que hace lo siguiente: primero llama a ADC_Read(PA0) para obtener la lectura del potenciómetro luego convierte ese valor a un rango de 0–100 % con duty. Después limpia la pantalla (LCD_Clear()), imprime el texto y el porcentaje con LCD_Print() y, finalmente, envía el valor por UART con UART_Write(USART1, duty). Para espaciar las lecturas y respetar el periodo de muestreo, al final del bucle se invoca un retardo por Delay_ms().

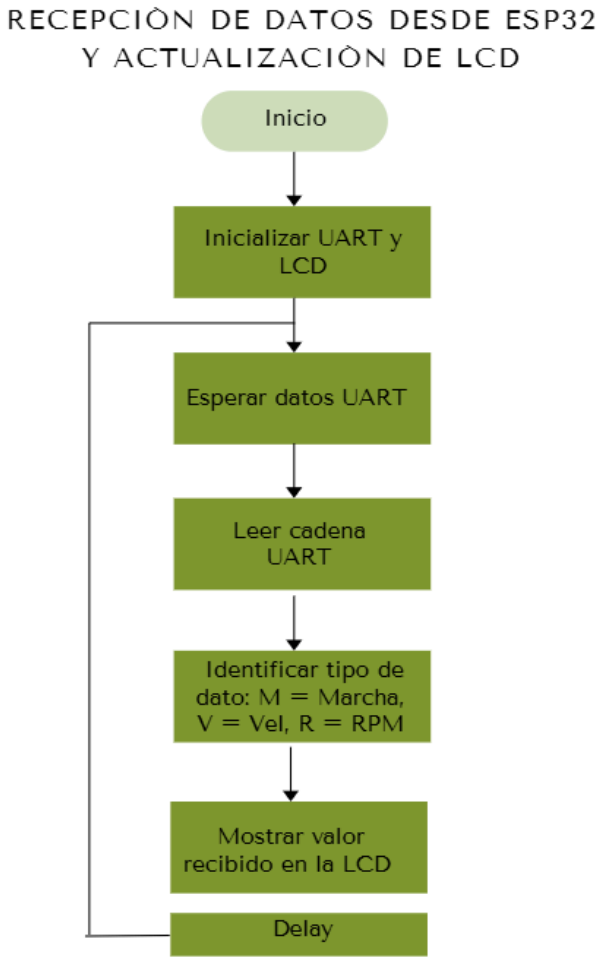


Figura 11. Recepción de datos

En este diagrama que es la figura 7, es la STM32 que recibe datos desde la ESP32 mediante comunicación UART. Estos datos incluyen información como la marcha, la velocidad del motor o las RPM. Cada valor es identificado, procesado y visualizado en tiempo real en el display LCD finalmente llama a Delay_ms() para respetar el periodo de muereo antes de repetir el ciclo.

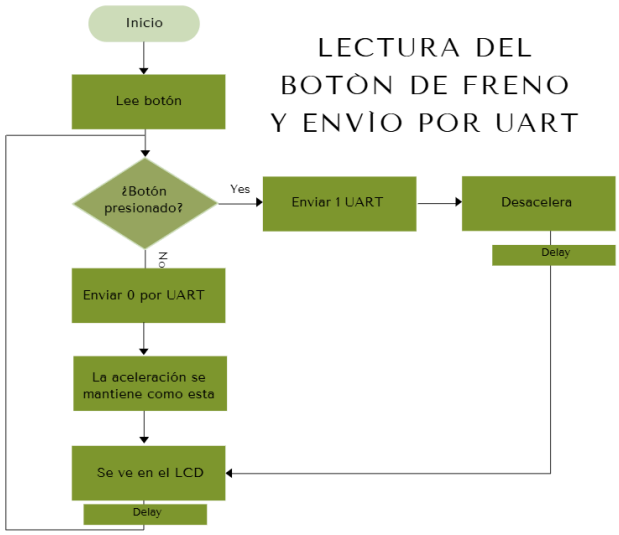


Figura 12. Lectura del botón

Al iniciar se configura el GPIO del botón con GPIO_Init(BUTTON_PA7) y el UART (UART_Init(USART1,115200)), luego el sistema entra en un bucle continuo que sigue este flujo: primero ejecuta state = GPIO_Read(BUTTON_PA7) para saber si el freno está presionado; si state == PRESSED, llama a UART_Write(USART1, 1) y a continuación invoca Decelerate() para ajustar el modelo de velocidad en la ESP32. En caso contrario, hace UART_Write(USART1, 0) y mantiene la velocidad actual sin cambios. Después, refleja el estado en la pantalla LDC y finalmente llama a Delay_ms() para respetar el periodo de muereo antes de repetir el ciclo.

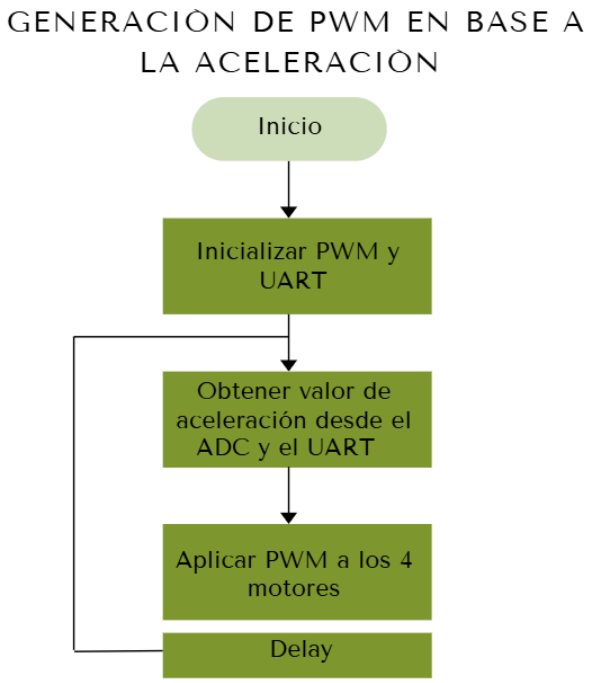


Figura 13. Generación de PWM

Este diagrama de la figura 9, describe la lógica que utiliza la STM32 para generar señales PWM que controla la intensidad de los cuatro motores, representando así visualmente la velocidad del vehículo y que se vea reflejado en los LEDs finalmente llama a Delay_ms() para respetar el periodo de muestreo antes de repetir el ciclo.

3.2.6.Sistema Operativo en Tiempo Real (RTOS)

Diagrama de interacción de las Tareas, a nivel del sistema

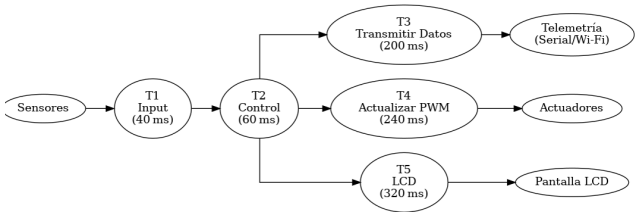


Figura 14. Diagrama de interacción

El diagrama muestra el flujo de datos en el sistema: los sensores alimentan a T1 (Input), que entrega la lectura a T2 (Control). T2 envía comandos de ciclo útil a T4 (Actualizar PWM) para accionar los actuadores, transmite telemetría a T3 para la salida por el canal de comunicación y actualiza el estado que T5 (LCD) presenta en la pantalla.

Tabla del set de Tareas: Periodos, Tiempos de Ejecución, Prioridades.

	Tarea	Periodo	Prioridad	Tiempos de Ejecucion
T1	Input	40 ms	1 (Mayor)	1 ms
T2	Control	60 ms	2	2 ms
T3	Transmitir Datos	200 ms	3	4 ms
T4	Actualizar del PWM	240 ms	4	4 ms
T5	LCD	320 ms	5 (Menor)	5 ms

En nuestro diseño RTOS se programaron cinco tareas periódicas que cubren todo el ciclo de operación del sistema, asignándoles la prioridad inversamente proporcional a su periodo para garantizar la respuesta en tiempo real. T1 (Input) se dispara cada 40 ms (1 ms de cómputo) y, al ser la encargada de muestrear los sensores de entrada, recibe la prioridad 1, que es la más alta. T2 (Control) corre cada 60 ms (2 ms), calcula la ley de control y por eso se sitúa inmediatamente después en criticidad. T3 (Transmitir Datos) envía telemetría al exterior cada 200 ms, consumiendo 4 ms su mayor periodo le permite la prioridad 3. T4 (Actualizar el PWM) refresca las salidas de potencia cada 240 ms (4 ms) para mantener una buena señal por lo que se ubica en la prioridad 4. Por último, T5 (LCD) actualiza la interfaz de usuario cada 320 ms (5 ms) y, al ser la menos crítica queda con prioridad 5.

Scheduling de las Tareas.

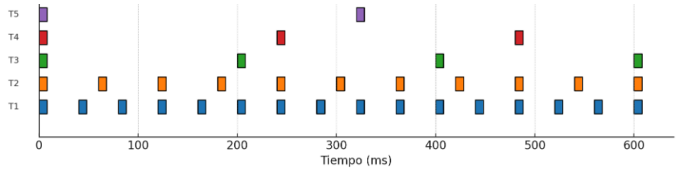


Figura 15. Scheduling de las Tareas.

El cronograma muestra cómo el planificador ejecuta las cinco tareas durante dos ciclos completos. Cada bloque coloreado representa una activación las cuales son:

- T1 azul se dispara cada 40 ms, por eso aparecen 16 bloques es la de mayor frecuencia.
- T2 naranja corre cada 60 ms, intercalada entre las ejecuciones de T1.
- T3 verde se activa a los 200 ms y 600 ms.
- T4 rojo lo hace a los 240 ms y 480 ms.
- T5 morado solo aparece una vez, al inicio del segundo ciclo 320 ms.

3.2.7.Interfaz de Usuario

Se implementó Matplotlib como interfaz gráfica principal por las siguientes razones:

- Simplicidad: Permite crear elementos interactivos (sliders, botones, gráficos) con pocas líneas de código.
- Integración con Matplotlib: Facilita la visualización en tiempo real de datos mediante su compatibilidad con librerías de gráficos.
- Multiplataforma: Funciona en varios entornos sin modificaciones adicionales.

La interfaz desarrollada incluye los siguientes componentes para el control del tractor:

Botones de acción:

- Inicio/Detención: Controla la simulación en tiempo real.
- Salida: Cierra la aplicación de forma segura.

Visualización en tiempo real:

- Gráfico dinámico: Muestra la evolución de las RPM en función del tiempo.
- Indicador numérico: Muestra el valor actual de RPM calculado.

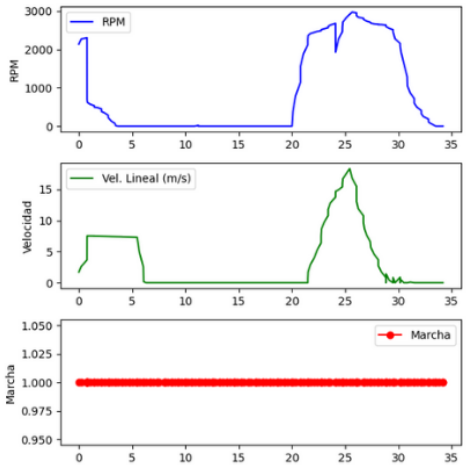


Figura 14. Interfaz gráfica

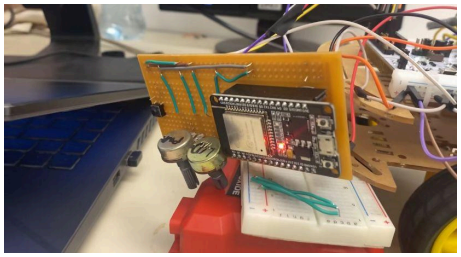
El usuario ajusta el freno (0-100%) para simular resistencia mecánica en el tractor. Al aumentar el freno, el modelo matemático reduce proporcionalmente la velocidad angular base, considerando la diferencia entre el torque aplicado y la fuerza de frenado. Esta reducción se traduce en una disminución de las RPM calculadas, las cuales se actualizan en tiempo real en el gráfico dinámico y el indicador numérico. Los valores de freno, velocidad angular y RPM se almacenan automáticamente en el archivo CSV, registrando el impacto de la acción. La interfaz garantiza coherencia entre los controles, los cálculos del modelo físico y la visualización, demostrando cómo el freno modula el comportamiento del sistema de manera predecible y cuantificable.

3.3.Pruebas de funcionamiento del prototipo

Lectura del botón para activar función específica

```
uint16_t pot_value = ADC_Read_Value();

printf("{adc: %u, button: %u}\n", pot_value, btn_flag);
```



Acción: Prueba de lectura de botón conectado en PA7.

Descripción:

- El botón se presiona o se suelta.
- El pin PA7 es leído en el ciclo principal y su valor se guarda en btn_flag.
- El valor se imprime, indicando si está presionado o no.

Relación con el código:

- Esta línea detecta si el pin PA7 está en alto (presionado).
- Permite implementar acciones condicionales basadas en la interacción del usuario (como encender un LED, cambiar de modo, etc.).

Comunicacion USART

```
void USART1_IRQHandler(void)
{
    if (USART1->ISR & (1U << 5))  // Check RXNE flag (bit 5)
  {
     char incoming = (char)USART1->RDR;

     if (incoming == 'V')  // Speed data
     {
        current_speed = atoi(str_temp);
                    memcpy(str_speed, str_temp, sizeof(str_temp));
        memset(str_temp, 0, sizeof(str_temp));
        char_count = 0;
     }
     else if (incoming == 'S')  // RPM data
     {
                    memcpy(str_rpm, str_temp, sizeof(str_temp));
        memset(str_temp, 0, sizeof(str_temp));
        char_count = 0;
     }
     else if (incoming == 'E')  // Gear (march) data
     {
                    memcpy(str_gear, str_temp, sizeof(str_temp));
        memset(str_temp, 0, sizeof(str_temp));
        char_count = 0;
     }
     else
     {
        if (char_count < MAX_BUFFER - 1)
        {
           str_temp[char_count++] = incoming;
        }
        else
        {
              // Buffer overflow: reset temporary buffer
           memset(str_temp, 0, sizeof(str_temp));
           char_count = 0;
        }
     }
  }
}
```

Acción: Representación de la aceleración del tractor en el LCD.

Descripción:

- La Raspberry Pi envía datos de velocidad al STM32 vía UART.
- STM32 los recibe, los almacena en current_speed, y llama a

update_motor_speed() para reflejar el cambio.
- La velocidad también se muestra en la pantalla LCD en tiempo real.

Relación con el código:

- El primer fragmento recibe la velocidad como cadena, la convierte a entero y actualiza current_speed.
- La función update_motor_speed() probablemente ajusta el PWM del motor según la velocidad.
- Esto simula la aceleración del tractor de manera visual y funcional.

Lectura potenciómetro

uint16_t pot_value = ADC_Read_Value();

Acción: Verificación de lectura del sensor analógico (potenciómetro).

Descripción:

- El usuario gira el potenciómetro.
- El STM32 lee el valor mediante ADC y lo muestra por consola vía printf.
- Este valor puede usarse para controlar otros parámetros como velocidad, brillo, etc.

Relación con el código:

- ADC_Read_Value() obtiene el valor analógico (de 0 a 4095 por ejemplo).
- Este valor es mostrado en consola y puede usarse en lógica interna del prototipo.
- Confirma que el sistema lee correctamente entradas analógicas.

3.4.Infraestructura

3.4.1. Sistema (STM32+Raspberry Pi)

• Descripción y diagrama(s) ilustrativo(s) del sistema.

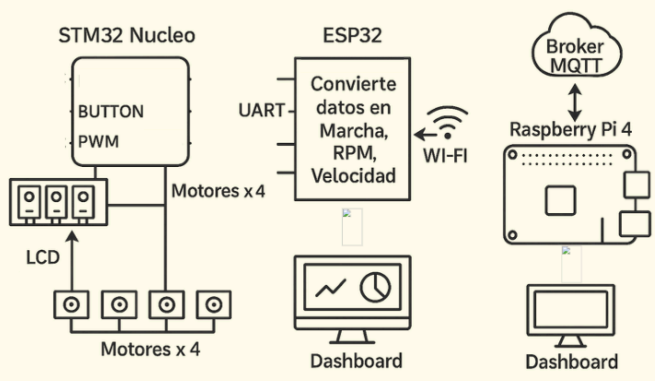


Figura 15. Diagrama ilustrativo

El sistema integra control embebido y monitoreo a través de cuatro partes clave. La STM32 Nucleo gobierna los cuatro motores con PWM, recibe la señal de un botón, potenciómetro o dashboard y muestra estados en un LCD. Mediante UART, el STM32 transmite a la ESP32 los parámetros de marcha, RPM y velocidad y la ESP32 los transforma para mandarlo como mensajes MQTT y los envía por Wi-Fi.

Una Raspberry Pi 4 actúa como broker MQTT y se encarga de almacenar los datos y alimenta un dashboard que permite visualizar en tiempo real la velocidad, aceleración y el PWM de los motores. Desde el dashboard se puede acelerar o controlar la velocidad de los motores también frenar y todo se verá graficado. Cuando se manda desde el dashboard. Lo manda a la ESP32 y la transforma para que se la pase a la STM32 y se vea reflejado el cambio en los motores y la LCD.

3.4.2 Recursos utilizados (materiales, equipos, entre otros

• Hardware y Prototipo. Lista detallada de los componentes físicos y electrónicos utilizados.

STM32 NUCLEO-G031C6: Microcontrolador principal que ejecuta FreeRTOS, lee el potenciómetro y el botón de freno, genera las señales PWM para los motores y se comunica por UART con la ESP32.

ESP32: Módulo Wi-Fi/Bluetooth que recibe la telemetría por UART, la convierte a marcha,

RPM y velocidad y la convierte en MQTT hacia la Raspberry Pi.

Raspberry Pi: corre el script Python que almacena los datos en CSV y alimenta el dashboard de visualización remota.

PCB: Tarjeta a medida que agrupa reguladores, headers y rutas de señal para simplificar el cableado entre STM32, ESP32, driver y periféricos.

Driver de motores TB6612FNG: Puente H dual que controla cuatro motores DC a partir de las salidas PWM del STM32.

Chasis acrílico con 4 motores DC y ruedas: Estructura mecánica que soporta las placas y proporciona tracción independiente a cada rueda.

Pantalla LCD: Muestra en tiempo real la marcha, las RPM, la velocidad y el porcentaje de acelerador.

Potenciómetro: Actúa como acelerador analógico; su lectura ADC define el duty cycle de los motores.

Botón: Entrada digital dedicada al frenado inmediato del vehículo.

Jumpers: Cables flexibles para las conexiones rápidas entre PCB y módulos.

Batería: Fuente de alimentación principal para motores y electrónica, con regulador step-down a 5 V para las placas de control.

• Software. Herramientas y plataformas de software empleadas para el desarrollo y simulación.

STM32 CubeIDE 1.15: Entorno de desarrollo todo-en-uno (editor, compilador) para generar el código C de la STM32, configurar FreeRTOS, drivers HAL/LL y trazar la ejecución en tiempo real.

Python 3.11: Lenguaje principal del backend en la Raspberry Pi materializa el pipeline de recepción MQTT, almacenamiento CSV y filtrado de datos.

Visual Studio Code: Editor multipropósito con extensiones de C/C++ y Python empleado para scripts, documentación y diagrama rápidos.

GitHub: Control de versiones distribuido y repositorio central para código fuente, diagramas PlantUML y reportes colaborativos.

Mosquitto 2.0: Servicio instalado en la Raspberry Pi que intermedia todos los mensajes publish/subscribe entre la ESP32 y las aplicaciones Python.

4.Resultados

4.1.Evaluación e interpretación de los resultados obtenidos.

• Referentes a la dinámica del sistema de conducción inteligente

(Interacción del STM32 con la Raspberry Pi, comunicación de datos, y desempeño de los sensores y actuadores en el sistema).

En la dinámica general del sistema de conducción inteligente, la interacción entre la STM32 y la Raspberry Pi ha mostrado un comportamiento de gran utilidad en la parte del envío periódico de mensajes MQTT desde el ESP32 que tiene el papel de intermediario entre STM32 y broker Mosquitto en la Rasp que permite que la Raspberry Pi actualice los dashboards en tiempo real. La capacidad de la Rasp para actuar como broker y servidor de visualización no interfiere con el lazo de control de la STM32 pues el microcontrolador trabaja de forma determinista gracias al esquema con el que se diseñó.

En cuanto a la comunicación de datos, el enlace UART tiene un baud rate de 115200 ya que se realizó los cálculos y era lo más eficiente para transmitir valores de aceleración, freno y RPM. El uso de interrupciones en la STM32 y de callbacks en el ESP32 garantiza que ni la lectura de sensores ni la generación de PWM se retrasen por la cola de transmisión.

El desempeño de los sensores que viene siendo el potenciómetro y botón de freno ha sido han sido los encargados de controlar la velocidad del tractor. La detección del freno por GPIO responde en menos de 2 ms desde la pulsación,

tiempo que incluye debounce software, por lo cual la respuesta de deceleración se siente instantánea desde la perspectiva del usuario.

Respecto a los actuadores, el control PWM de los cuatro motores con TIM1 a 1 kHz ofrece una regulación suave de velocidad sin perceptibles parpadeos ni ondulación mecánica. Las curvas de aceleración y desaceleración experimentales coinciden con las expectativas del modelo de simulación la transición completa del 0 % al 100 % de duty-cycle tarda aproximadamente 40 ms, lo que corresponde a un cambio gradual y seguro en la aceleración.

IV. Conclusiones

En síntesis, esta etapa final demostró la integración efectiva de todos los conceptos abordados en el curso, culminando en un sistema donde motores, botón de freno, potenciómetro y pantalla LCD conviven bajo la coordinación determinista de FreeRTOS en la STM32. La comunicación bidireccional UART entre STM32 y ESP32, y MQTT entre ESP32 y Raspberry Pi permitió un flujo continuo de datos: el microcontrolador envía la telemetría, la ESP32 la traduce y la Pi la almacena y grafica en un dashboard que, a su vez, ofrece control remoto de aceleración y frenado. El desarrollo en C garantizó la respuesta en tiempo real de los periféricos críticos, mientras que el script en Python amplió la visibilidad operativa con registro en CSV y visualización dinámica. Pese a los desafíos especialmente la correcta configuración del RTOS y la sincronización de tareas cumplimos los objetivos del reto y validamos la arquitectura propuesta. Como trabajo futuro, planeamos diseñar una PCB dedicada y mejorar la estética y la ergonomía del chasis para acercarnos aún más a la apariencia y robustez de un tractor agrícola real.

V. Bibliografía

[1] Deere, J. (2025). Autonomous Tractor | John Deere. Deere.com. https://www.deere.com/en/autonomous/

[2] Learn The Role Of Embedded Systems In The Automotive Industry. (2024, September 27). Sunstream. https://www.sunstreamglobal.com/learn-the-role-of-embedded-systems-in-the-automotive-industry/

[3] Agricultura mundial: hacia los años 2015/2030. (2015). Fao.org. https://www.fao.org/4/y3557s/y3557s08.htm

[4] Sankar, G. (2022, March 7). How embedded vision is contributing to the AI revolution in autonomous tractors. E-Con Systems. https://www.e-consystems.com/blog/camera/applications/how-embedded-vision-is-contributing-to-the-ai-revolution-in-autonomous-tractors/

[5] Soni, A. (2023, October 31). What is a System on a Chip (SoC)? Ansys.com; Ansys Inc. https://www.ansys.com/blog/what-is-system-on-a-chip

[6] IBM. (2024, June 4). Microcontrolador. Ibm.com. https://www.ibm.com/mx-es/think/topics/microcontroller#:~:text=Schneider%2C%20Ian%20Smalley-,%C2%BFQu%C3%A9%20es%20un%20microcontrolador?,requerir%20un%20sistema%20operativo%20complejo.

[7] ¿Qué es un system on a chip? (2020). /Clases/1098-Ingenieria/6552-Que-Es-Un-System-On-a-Chip/. https://platzi.com/clases/1098-ingenieria/6552-que-es-un-system-on-a-chip/

[8] Gillis, A. S., & Lutkevich, B. (2024). What is an embedded system? Search IoT; TechTarget. https://www.techtarget.com/iotagenda/definition/embedded-system

[9] Real-Time Operating System (RTOS): Working and Examples | Spiceworks. (2024, March 7). Spiceworks Inc; Spiceworks. https://www.spiceworks.com/tech/hardware/articles/what-is-rtos/

[10] RTOS vs Bare-Metal. (2023). Translate.goog. https://www-embeddedrelated-com.translate.goog/thread/5762/rtos-vs-bare-metal?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=sge

[11] lolate.com, & TRBL. (2021, July 27). LINUX Embebido | Qué es, cómo funciona y para qué se usa. TRBL Services. https://trbl-services.eu/blog-linux-embebido-que-es/

[12] Evolute Group. (2025, March 26). Evolute - Evolute. https://www.evolute.in/blog/how-embedded-operating-systems-are-taking-over-iot-and-automation/

[13]Embedded Linux. (2020, April 22). Critical Link. https://www.criticallink.com/embedded-linux/

[14] A novel embedded system for tractor implement performance mapping. (2024). Cogent Engineering. https://doi.org/10.1080//23311916.2024.2311093

[15] Sankar, G. (2022, March 7). How embedded vision is contributing to the AI revolution in autonomous tractors. E-Con Systems. https://www.e-consystems.com/blog/camera/applications/how-embedded-vision-is-contributing-to-the-ai-revolution-in-autonomous-tractors/

[16] lolate.com, & TRBL. (2020, November 3). Sistema embebido: C/C++ para el desarrollo de software embebido. TRBL Services. https://trbl-services.eu/blog-que-es-lenguaje-c-donde-se-aplica/

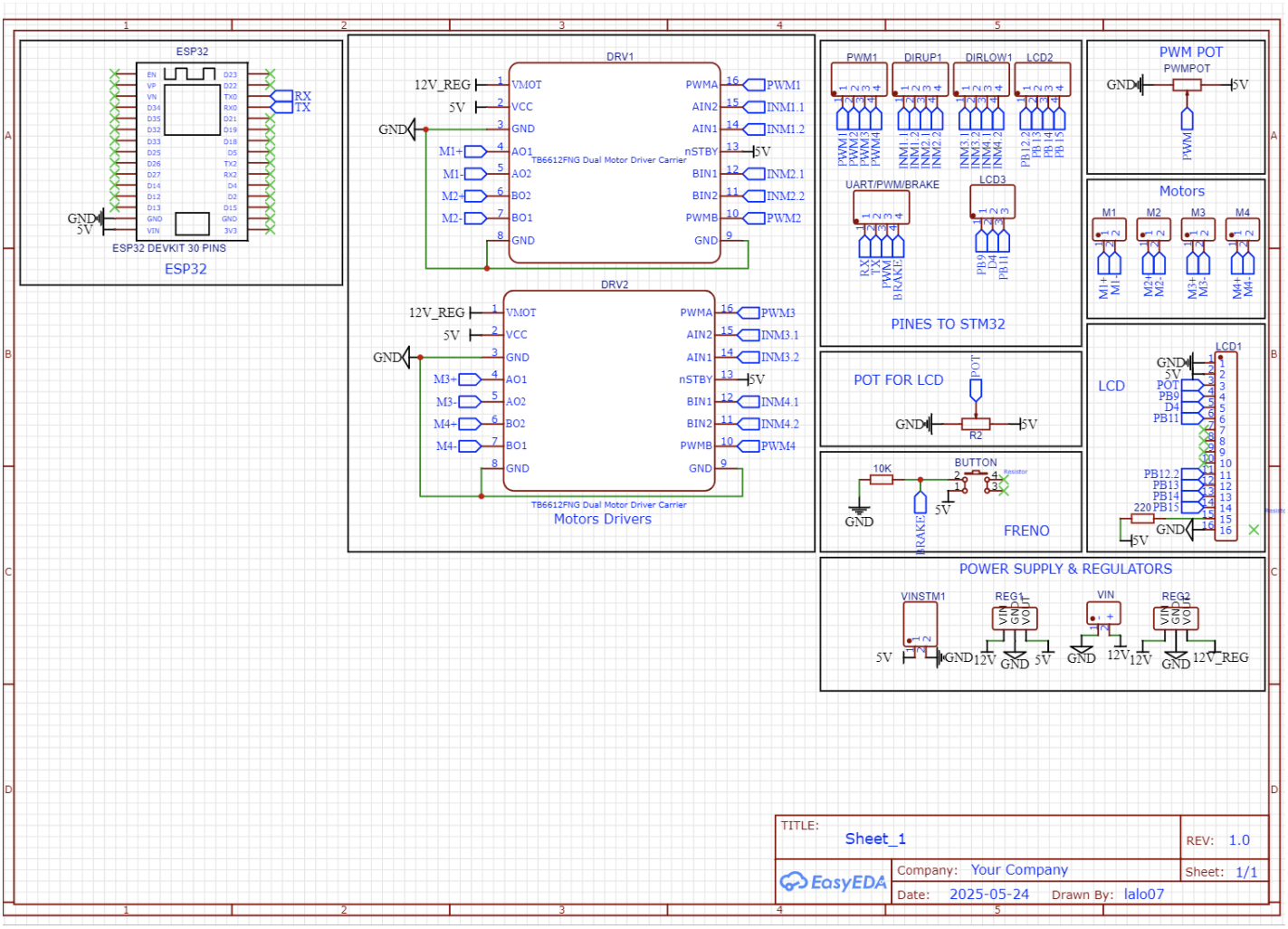
[17] Rohde. (2025). Qué es UART. Rohde-Schwarz.com. https://www.rohde-schwarz.com/es/productos/test-y-medida/essentials-test-equipment/digital-oscilloscopes/que-es-uart_254524.html

[18] ¿Qué es el MQTT? - Explicación del protocolo MQTT - AWS. (2022). Amazon Web Services, Inc. https://aws.amazon.com/es/what-is/mqtt/

ANEXOS

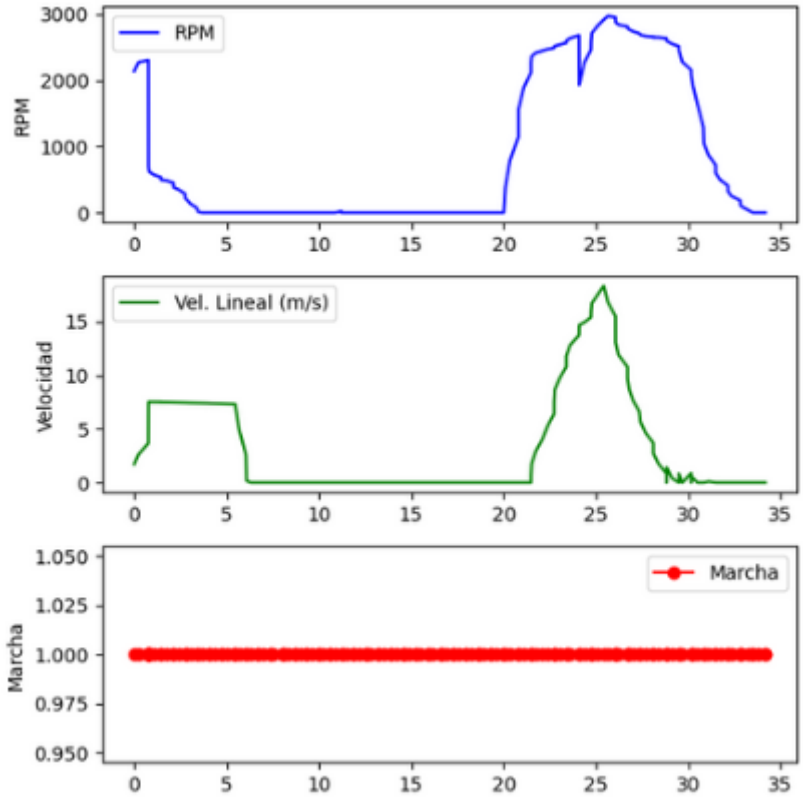
Nota: Dado que VS Code es un editor relativamente pesado, y no se tenía certeza sobre cómo sería clonado nuestro archivo desde un repositorio de GitHub, se recurrió a ChatGPT para obtener orientación sobre los pasos a seguir. Esto se debió a que, en otra Raspberry Pi, al intentar instalar VS Code, la aplicación se cerraba por un error.

Anexo 1. Repositorio de Github: https://github.com/htovarm7/SoC



Anexo 2. Diagrama esquemático

Anexo 3. Esquemático y placa: https://easyeda.com/editor#project_id=e10f5c8423ef438f951b31ac4fa864ba



Anexo 4. Graficas