Documentación de la Implementación de RA-TDMAp para RUPU

Trabajo para Título en Ingeniería Civil Eléctrica
para Pontificia Universidad Católica de Chile

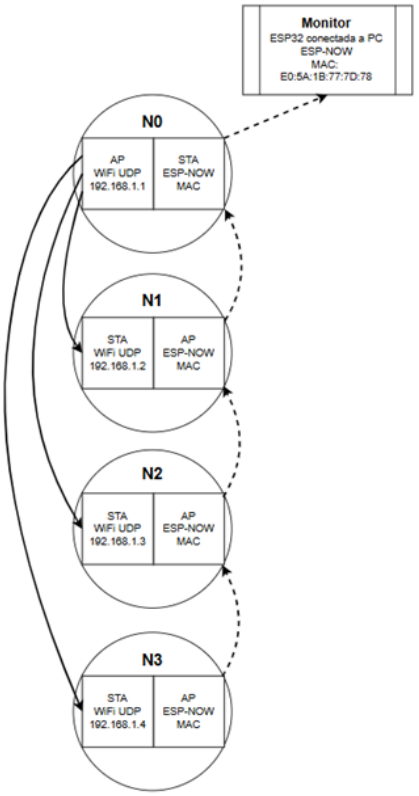
2025

Santiago Voticky

Formación de Pelotón:

El archivo secuencia_orden.ino se encarga de implementar la fase de *formación de pelotón*. Su objetivo es descubrir y ordenar a los robots que participarán en la red, asignando a cada uno una posición única. Para ello, el líder levanta un punto de acceso WiFi y comienza a recibir por UDP broadcast las direcciones MAC que envía cada seguidor. Estas direcciones se van guardando en el vector macList, evitando duplicados gracias a la función macExists(). La posición de cada dirección en la lista puede consultarse con findMacIndex(). El líder mantiene un conteo del número de estaciones vistas y, cuando este coincide con WiFi.AP.stationCount(), que representa el número de clientes conectados al AP, difunde por UDP el mensaje "Todos listos". A partir de ese momento, la lista se congela y ya no se aceptan nuevas MAC. Cada robot, incluyendo el líder, obtiene su propia dirección con WiFi.softAPmacAddress() o con la dirección STA según corresponda, la busca en la lista y determina su índice. Ese valor se guarda en la variable contador_estaciones, que define la posición del robot en el pelotón, mientras que cantidad_peloton almacena el número total de participantes. Además, la variable estoy_ordenado se marca en 1 para indicar que el proceso terminó correctamente. Si el robot tenía activa alguna tarea de movimiento, representada por movimiento_handle, esta se elimina con vTaskDelete() y los motores se detienen llamando a motor(0,0). De este modo, el sistema asegura que todos los robots conozcan su lugar dentro de la formación antes de avanzar a la siguiente etapa. Cabe mencionar que si un robot se conecta tarde, después de que el líder cerró la fase, su MAC no aparecerá en la lista, por lo que será necesario repetir el proceso. En resumen, este módulo establece la base organizativa del pelotón y garantiza que la posición de cada miembro sea conocida y única.

La topología que se forma para 4 robots se puede ver en la figura.



Sincronización:

Por otro lado, el archivo sincronizacion.ino se ocupa de alinear los relojes internos de todos los robots, lo que resulta esencial para que funcione el esquema de comunicación *TDMA*. La variable central en este proceso es desfase, que representa el offset que cada robot debe sumar a su función micros() para obtener un reloj ajustado y común a todo el sistema. El estado general de la sincronización se controla con sincronizado, que inicia en 0 y pasa a 1 cuando todos los nodos quedan alineados. Existen también banderas de control como estado_sync, estado_seguidor y sync_iniciada, que determinan si el intercambio de tiempos está en curso o ya finalizó, además de la variable contador_sincronizados, que lleva la cuenta de los robots que han confirmado la sincronización. El intercambio de tiempos se apoya en variables como time_past y time_now, que marcan los instantes de recepción local, junto con time_other_1 y time_other_2, que son tiempos enviados por el líder. A partir de ellos, el seguidor calcula los retardos de ida y vuelta (round_trip_1 y round_trip_2) y obtiene un valor promedio para compensar posibles asimetrías de red. Con estos valores, el desfase se estima mediante la fórmula desfase = time_other_2 - time_now + (round_trip_2 / 2), lo que aproxima la diferencia de relojes. Se evalúa además un valor de predicción time_prediction, y si la diferencia con el tiempo enviado por el líder es menor a 100 microsegundos, se considera que el robot quedó sincronizado. En ese caso, el seguidor envía varias confirmaciones de "OK" y suspende la tarea udp_sync_task, creada inicialmente en el líder para manejar el proceso. El líder, por su parte, incrementa contador_sincronizados con cada confirmación recibida y, una vez que el número de sincronizados coincide con el total de estaciones conectadas, difunde el mensaje "Todos estan sincronizados" y marca la variable global sincronizado en 1.

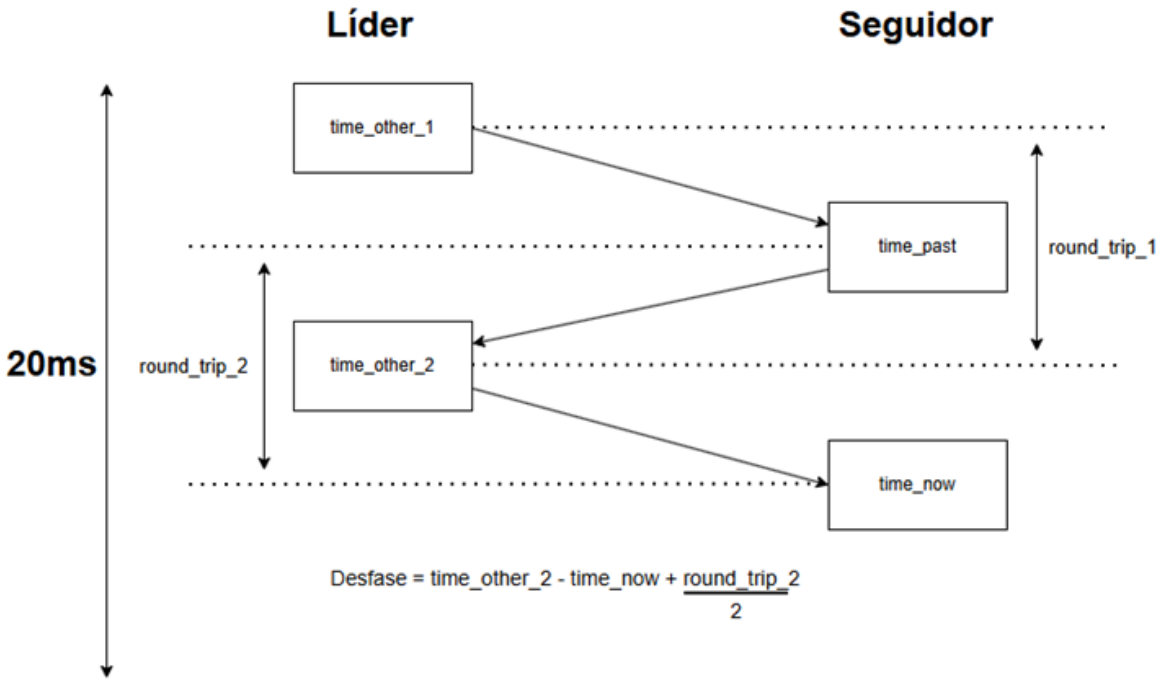
Con el valor de desfase cada robot puede calcular un reloj ajustado como time_adjusted = micros() + desfase, lo cual permite definir sus intervalos de transmisión TDMA (mi_inicio_TDMA y mi_final_TDMA) de manera coherente con el resto del sistema. Es importante destacar algunas consideraciones: el umbral de ±100 microsegundos equilibra la precisión con las limitaciones de latencia y jitter propias de UDP; el uso de dos retardos promedio reduce los sesgos causados por retardos asimétricos; el hecho de que el seguidor envíe múltiples mensajes "OK" mejora la robustez frente a pérdidas de paquetes; y si la sincronización falla, la bandera sync_iniciada se reinicia a 0, quedando el líder listo para un nuevo intento.

Ambos módulos se integran dentro de la máquina de estados definida en el archivo principal RUPU_TDMA_sentido_paper_V2.ino. La secuencia general pasa de Inicio a Formacion_de_Peloton, luego a Sincronizacion, y posteriormente a Calibracion y Loop_de_Control. Las variables contador_estaciones y cantidad_peloton producidas por el módulo de formación, junto con desfase y sincronizado generados por el módulo de sincronización, son fundamentales para calcular los slots TDMA y asegurar que la comunicación por ESP-NOW se realice de manera ordenada y sin colisiones. En caso de necesitar cambios, se puede modificar el criterio de ordenamiento del pelotón ajustando cómo se llena macList o implementando un ordenamiento extra por RSSI o ID. Asimismo, la precisión de la sincronización se puede aumentar ajustando el umbral de error o

incrementando el número de ciclos de intercambio antes de confirmar convergencia. También es posible reforzar la tolerancia a fallos añadiendo temporizadores para reiniciar la tarea de sincronización si no hay progreso o repitiendo periódicamente los mensajes de confirmación.

En conjunto, la formación de pelotón y la sincronización de tiempo constituyen las fases preparatorias que aseguran que los robots estén ordenados y alineados en el tiempo, lo que habilita la ejecución posterior del esquema TDMA con precisión y confiabilidad.

El flujo de una ronda de sincronización se puede ver en la siguiente figura:



ESP-NOW:

La mensajería de bajo nivel entre robots se realiza con ESP-NOW y se orquesta por encima con un esquema TDMA. El intercambio de paquetes no es continuo ni por sondeo: cada paquete recibido se maneja vía el *callback* OnDataRecv, que se registra al inicializar ESP-NOW. Ese *callback* no ejecuta lógica pesada ni de control en caliente; en su lugar, copia el paquete entrante a una estructura local del tipo struct_message y la encola en qMsg mediante xQueueSend sin bloqueo. Esta decisión de diseño es clave: evita trabajo largo dentro de la ISR/callback y garantiza que la lógica de aplicación (control, actualización de estados, cómputos de TDMA) se ejecute después, en un contexto seguro, fuera del slot de transmisión del nodo. La cola qMsg se crea con xQueueCreate(10, sizeof(struct_message)), por lo que mantiene hasta diez mensajes pendientes de procesar; más adelante, en el bucle de control, una función de drenaje (por ejemplo, pollEspNowQueue) hace xQueueReceive repetidamente, consume los mensajes y llama a processEspNowMsg para aplicar los cambios necesarios. Esta fase de drenaje se invoca deliberadamente cuando el nodo no está en su

ventana de transmisión TDMA, de forma que el procesamiento no compita con el envío de paquetes del propio robot.

La estructura de datos que viaja por ESP-NOW está definida como struct_message y sirve tanto para datos de control del ciclo TDMA como para telemetría de movimiento. Tiene un campo tipo_mensaje (carácter) que distingue el contenido lógico del paquete; en tu implementación se usa, por ejemplo, 'M' para mensajes de mando/estado que el líder difunde al inicio de cada ronda. Para permitir empaquetar series cortas por ronda, muchos campos están definidos como arreglos de longitud MAX_LISTAS (en tu proyecto #define MAX_LISTAS 4). Así, enviados[] lleva un recuento local de mensajes emitidos, recibidos_m[] y recibidos_v[] contabilizan paquetes “M” y “V” recibidos según tu rol, atiempo[] guarda marcas de tiempo internas útiles para depuración o validación temporal, mientras que vel_1[] y vel_2[] almacenan magnitudes cinemáticas (p. ej., velocidad medida y referencia) y curva[] expresa la curvatura o set-point lateral proveniente del predecesor. Además, el arreglo time_stamp[] guarda sellos temporales en microsegundos y se utiliza para anclar datos a la línea temporal común (ajustada con el desfase obtenido en sincronización). Por último, datos_TDMA[4] transporta la metainformación de la ronda: índice 0 es cantidad_peloton, índice 1 es duracion_ronda_TDMA (en microsegundos), índice 2 es inicio_siguiente_ronda (marca temporal absoluta en la escala común), e índice 3 lleva rondas_tdma (un contador de rondas útil para coherencia y diagnóstico). El emisor habitual de estos cuatro parámetros es el líder, y los seguidores los consumen para calcular su slot_TDMA y los instantes mi_inicio_TDMA y mi_final_TDMA usando su contador_estaciones obtenido en la formación del pelotón.

En el líder, el flujo típico de envío ocurre dentro de su propio slot. Antes de transmitir, incrementa un contador local mensajes_enviados y rellena myData (instancia global de struct_message) con la información relevante de esa iteración: escribe tipo_mensaje = 'M' al comienzo de la ronda, copia el estado de paro en el campo para (por ejemplo “si”/“no”), coloca en vel_1[0] y vel_2[0] la velocidad medida y la referencia (Input_vel y vel_ref), en curva[0] la curvatura actual, y completa datos_TDMA[0..3] con el tamaño del pelotón, la duración de la ronda, el inicio absoluto de la siguiente ronda y el contador de rondas. Con esa estructura lista, el líder difunde el paquete mediante esp_now_send a mac_broadcast (FF:FF:FF:FF:FF:FF), de modo que todos los seguidores reciban exactamente la misma vista del ciclo de TDMA. A nivel de configuración de enlaces, el líder registra como *peers* la broadcast y, en tu código, también un mac_monitor para depuración; el *callback* de envío puede registrarse (está comentado) para confirmar entregas, pero la lógica de aplicación no depende de esa confirmación para avanzar. El diseño asume que, si un paquete de sincronización se pierde, habrá otro en la próxima iteración o en la siguiente ronda, y los seguidores mantienen su planificador con los parámetros más recientes que les hayan llegado.

En los seguidores, el rol es esencialmente reactivo y temporizado: escuchan paquetes struct_message del líder, los reciben vía OnDataRecv y los colocan en la cola qMsg junto con un time_stamp local ya ajustado (en el *callback* se hace msg.time_stamp[contador_estaciones - 1] = micros() + desfase). La razón de sumarle el desfase en este punto es fijar cada mensaje

entrante a la escala temporal común en el instante exacto de su llegada, dejando lista esa marca para lógicas posteriores de validación o estimación de retardo efectivo. Más tarde, cuando el seguidor no está en su ventana de transmisión, el lazo de control llama a la rutina de drenaje (por ejemplo pollEspNowQueue) con un pequeño presupuesto temporal y/o límite de ítems; esa rutina hace xQueueReceive repetidamente y, por cada elemento, invoca processEspNowMsg. Dentro de processEspNowMsg se distinguen los tipos de mensaje: cuando llega un paquete 'M' desde el predecesor o el líder, el seguidor actualiza variables de control como la saturación longitudinal (sat_d = msg.vel_2[i2]) y la curvatura_predecesor = msg.curva[i2] utilizando índices relativos a su posición (i1 = contador_estaciones - 1 para sí mismo e i2 = contador_estaciones - 2 para el predecesor). Cuando el mensaje es de tipo 'V', actualiza contadores de recibido, estampa en su propio índice el tiempo relativo a inicio_predecesor_TDMA, y puede, si así lo decides, refrescar atiempo[] con la hora ajustada. Ese diferimiento del cómputo —recibir en *callback* y procesar más tarde en lazo— asegura que el cálculo y la aplicación de las consignas no interfieran con la disciplina temporal del TDMA.

La coordinación temporal entre ambos roles se apoya en los parámetros TDMA que viajan en datos_TDMA[]. Tras haber corrido la sincronización NTP-like por UDP y fijado el desfase, cada nodo mantiene un reloj común time_adjusted = micros() + desfase. El líder computa slot_TDMA = duracion_ronda_TDMA / cantidad_peloton y difunde el inicio_siguiente_ronda absoluto; con ello cada seguidor calcula sus ventanas exactas: mi_inicio_TDMA = inicio_ronda_TDMA + slot_TDMA * (contador_estaciones - 1) y mi_final_TDMA = mi_inicio_TDMA + slot_TDMA. Así, el seguidor solo transmite dentro de su slot (por ejemplo, reenviando su propio myData hacia el sucesor u otro destino definido por tu topología) y el resto del tiempo drena la cola qMsg, corre su controlador y escucha. Este patrón mantiene el canal libre de colisiones, porque el envío se restringe al intervalo asignado y el procesamiento de recepción se desplaza a las zonas fuera del slot.

En términos de estados y variables clave, conviene recordar que rango_robot define el rol (1 líder, 0 seguidor), contador_estaciones fija la posición en el pelotón y orienta qué índice de los arreglos corresponde a “yo” o a mi predecesor, desfase materializa el ajuste de reloj, y qMsg es el búfer FIFO que desacopla interrupciones de la lógica de control. Del lado de la estructura compartida struct_message, tipo_mensaje clasifica el paquete; enviados[], recibidos_m[] y recibidos_v[] sirven para telemetría y diagnóstico; para[3] acarrea la orden de paro; atiempo[], time_stamp[] y datos_TDMA[] sostienen la coherencia temporal; y vel_1[], vel_2[], curva[] transportan variables cinemáticas y de trayectoria. Con todo esto, la arquitectura logra que el *callback* de recepción sea mínimo y determinista, el procesamiento sea diferido y controlado, y el envío quede estrictamente acotado a los slots TDMA, lo que en conjunto proporciona una comunicación confiable y ordenada entre el líder y los seguidores.

Cambios en Encoders:

En RUPU_TDMA_sentido_paper_V2.ino incluyes explícitamente la librería y fuerzas el modo sin ISR internas con #define ENCODER_DO_NOT_USE_INTERRUPTS antes de #include <Encoder.h>. Luego creas dos objetos de la librería: Encoder encoder_der(5, 23); y Encoder encoder_izq(18, 19);, que conectan el encoder derecho a los pines 5 (canal A) y 23 (canal B), y el encoder izquierdo a 18 (A) y 19 (B). Aunque desactivas las ISR internas de la librería, tú mismo adjuntas interrupciones de cambio de nivel a esos cuatro pines con attachInterrupt(..., CHANGE), y en cada ISR (encoder1_ISR para el derecho y encoder2_ISR para el izquierdo) llamas a encoder_der.read() o encoder_izq.read(). Ese patrón mantiene actualizado el estado interno del objeto Encoder en cada flanco A/B, pero deja el cómputo pesado para cuando tú lo leas en el lazo de control. Es decir, la librería no instala sus propias ISR; tú gatillas lecturas "ligeras" en tus ISR y más tarde consultas el contador acumulado con read() desde tu lógica.

Cambios en analogWrite:

El archivo define primero analogWriteNormalized(pin, value, valueMaxUser), un envoltorio para mapear una consigna de usuario a la resolución real del PWM de la ESP32. Internamente calcula levels = (1 << pwmResolution_bits) (donde pwmResolution_bits es la resolución de LEDC que configuraste en la inicialización, por ejemplo 8, 10, 12 bits, etc.). Con ese dato obtiene el duty en cuentas del temporizador proporcional a value dentro del rango [0, valueMaxUser], saturando a levels-1 si el usuario se pasa del máximo. Luego llama a analogWrite(pin, duty). El objetivo es desacoplar el rango lógico que usas en el código (por ejemplo 0…1024) de la resolución física del PWM, de manera que si cambias pwmResolution_bits no tengas que reescalar todo el resto del proyecto. En términos prácticos: si decides que resolucionPWM sea 1024 pero tu LEDC está a 12 bits (4096 niveles), analogWriteNormalized hace la conversión correcta por ti.

Un detalle importante es que, incluso en el caso "cero", la función mantiene la coherencia entre dirección y magnitud: pone los dos pines de dirección en bajo y escribe PWM 0 mediante analogWriteNormalized, asegurando que no quede un duty residual en los canales LEDC. La combinación de saturación previa con el mapeo por analogWriteNormalized evita discontinuidades cuando cambias de resolución (por ejemplo, si pasas de 10 a 12 bits en PWM) y mantiene el rango lógico de las consignas estable para el resto del proyecto (resolucionPWM sigue siendo tu "máximo" de usuario).

ESP32 Monitor:

Este archivo implementa un monitor mínimo de ESP-NOW pensado para escuchar paquetes que usan tu estructura común struct_message, desacoplar su procesamiento con una cola FreeRTOS y, opcionalmente, registrar dinámicamente al líder como *peer* para poder enviarle algo si hiciera falta. Arranca incluyendo WiFi.h, esp_wifi.h y esp_now.h, y define un búfer mac_formateada[6] para almacenar la MAC del líder en formato binario. La bandera global peer_registrado controla si ya está registrado el *peer* (parte en 1, es decir, no vuelve a registrar a menos que la pongas a 0). También fija MAX_LISTAS = 4 y declara la estructura

struct_message idéntica a la que usas en el resto del proyecto: tipo_mensaje (char), contadores enviados[], recibidos_m[], recibidos_v[], la cadena corta para[3], arreglos temporales y cinemáticos (atiempo[], vel_1[], vel_2[], curva[], time_stamp[]) y, muy importante, datos_TDMA[4] para transportar metadatos de cada ronda. Junto a ella, el archivo declara un esp_now_peer_info_t peerInfo reutilizable y una cola QueueHandle_t qMsg para pasar mensajes del *callback* a una tarea de consumo.

Para formatear la dirección del líder, la función guardar_mac_lider(String mac_lider) recibe una MAC en forma de texto "AA:BB:CC:DD:EE:FF", la parsea con sscanf a seis enteros hexadecimales y los copia a mac_formateada (tipo uint8_t[6]). Esta función se invoca más adelante cuando llega el primer mensaje válido y aún no hay *peer* registrado.

El *callback* de envío OnDataSent está implementado solo a modo de depuración: imprime "Delivery Success/Fail" según el esp_now_send_status_t. No es crítico para el flujo del monitor, y de hecho el registro del callback está comentado en setup().

El corazón de la recepción es OnDataRecv(const uint8_t* mac, const uint8_t* incomingData, int len). Aquí no se hace trabajo pesado: se declara un struct_message incomingReadings local, se copia el paquete completo con memcpy(&incomingReadings, incomingData, sizeof(incomingReadings)) y se encola inmediato con xQueueSend(qMsg, &incomingReadings, 0). Este patrón es correcto para ESP-NOW: el *callback* es breve y no bloqueante, y toda la lógica de aplicación pasa a una tarea normal de FreeRTOS.

Esa lógica vive en consumerTask(void* pv). La tarea ejecuta un bucle infinito y bloquea en xQueueReceive(qMsg, &incomingReadings, portMAX_DELAY) esperando nuevos paquetes. Cuando recibe uno, primero filtra por tipo_mensaje (procesa cuando es 'M' o 'V', que en tu sistema representan control/telemetría). Si peer_registrado == 0, toma el campo incomingReadings.para (que en tu pipeline usas para transportar identificadores; aquí lo usas para derivar la MAC del líder), llama a guardar_mac_lider(...), carga peerInfo.peer_addr con esa MAC, setea peerInfo.channel = 0 y peerInfo.encrypt = false, y ejecuta esp_now_add_peer(...). Si el alta falla, imprime "Failed to add peer". Esto habilita que, si quisieras, respondas o envíes algo al líder más adelante. Tras esa rama de registro, la tarea continúa con el procesamiento del paquete: en tu archivo hay varias líneas de Serial.printf(...) (algunas comentadas) que muestran cómo depuras campos como datos_TDMA[0..3] y, típicamente, vas iterando índices i=0..MAX_LISTAS-1 para imprimir vectores como vel_1[], vel_2[], curva[], enviados[], recibidos_m[] y recibidos_v[]. La idea es que este monitor quede escuchando y mostrando los contenidos de cada struct_message que circula en la red, con orden FIFO garantizado por la cola.

La inicialización ocurre en setup(). Configuras el puerto serie a 115200, pones el Wi-Fi en modo estación con WiFi.mode(WIFI_STA), fuerzas el canal 1 con WiFi.setChannel(1) (esto es clave para que ESP-NOW funcione: todos deben estar en el mismo canal) y esperas a que la STA arranque. Si esp_now_init() devuelve error, lo informas y abortas. Luego (opcionalmente) no registras el callback de envío (está comentado) y sí registras el callback de recepción con esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv)). Finalmente,

creas la cola con xQueueCreate(10, sizeof(struct_message)) y levantas consumerTask pinneada al core 1 con xTaskCreatePinnedToCore(..., 1). El loop() queda prácticamente vacío (solo un delay(1000)), porque todo lo hace la combinación callback → cola → tarea.

En síntesis, monitor_espnow_mini_v1.ino es una herramienta de escucha y diagnóstico de tu tráfico ESP-NOW con la misma struct_message que usa tu red: recibe paquetes vía callback, los encola sin bloquear y los procesa ordenadamente en una tarea dedicada; puede registrar en caliente la MAC del líder como peer (si habilitas peer_registrado == 0) y deja trazas con Serial.printf de los campos más relevantes, incluidos los parámetros datos_TDMA[] de cada ronda. La arquitectura está pensada para ser robusta (poco trabajo en ISR), portable (mismo struct_message) y segura en tiempo (el consumo ocurre fuera de la interrupción), de modo que puedas auditar fácilmente qué se está enviando/recibiendo en tu sistema cooperativo.

Procesamiento de datos en Python:

Este script en Python está diseñado para leer y registrar los datos enviados por la ESP32 a través del puerto serie. Su objetivo principal es capturar la información que imprime el microcontrolador (por ejemplo, los paquetes TDMA o los valores de telemetría de cada ronda), darle un formato legible y guardarla en un archivo CSV para poder analizarla después con más comodidad.

El programa comienza importando las librerías necesarias: serial y time para comunicarse con el puerto serie, re para aplicar expresiones regulares al texto recibido, csv para escribir archivos, y además Path y datetime para gestionar rutas y nombres de archivo con marcas de tiempo. Define primero la configuración del puerto: la variable PUERTO contiene el nombre del puerto donde está conectada la ESP32 (por ejemplo "COM3" en Windows o "/dev/ttyUSB0" en Linux), mientras que BAUD fija la velocidad de comunicación (115200 baudios en tu caso). También crea la ruta del archivo de salida CSV_PATH, llamada normalmente datos_esp32.csv.

La parte más importante es la expresión regular definida en pat_linea. Esta se encarga de reconocer las líneas que imprime la ESP32 con el formato Datos[i] = valor1 ; valor2 ; valor3 ; .... El script busca exactamente ese patrón y, si encuentra una coincidencia, extrae el índice i y la lista de valores numéricos que siguen separados por punto y coma. Para organizar la información, defines una lista llamada CAMPOS con los nombres de cada columna: por ejemplo "ts", "idx", "enviados", "rx_m", "rx_v", "estado", "vel_1", "vel_2", "curva" y "delay". Esto asegura que cada dato tenga su encabezado al exportarse a CSV.

La función parsear_linea(linea: str) aplica la expresión regular a cada línea recibida. Si la línea coincide con el formato esperado, la función separa los valores por ;, los convierte a enteros o flotantes según corresponda y los devuelve en un diccionario con claves que

corresponden a los nombres de CAMPOS. Si la línea no coincide, devuelve None, ignorando así cualquier texto que no sean datos formateados.

En la rutina principal, el script abre el puerto serie usando serial.Serial(PUERTO, BAUD), espera un momento para estabilizar la conexión y luego entra en un bucle infinito leyendo línea por línea. Cada línea recibida se decodifica de bytes a string, se pasa a parsear_linea, y si efectivamente contiene datos válidos, se escribe en el archivo CSV con un escritor de la librería csv. De esta manera, todo el tráfico relevante que emite la ESP32 queda registrado cronológicamente. Para organizar los archivos, el script suele incluir la fecha y hora actual en el nombre de salida, garantizando que cada ejecución cree un archivo distinto y no sobrescriba registros anteriores.

En resumen, lector_datos.py es una herramienta de captura y almacenamiento: escucha el puerto serie donde la ESP32 imprime sus datos, filtra las líneas que tienen formato válido con una expresión regular, les asigna encabezados predefinidos y las guarda en un archivo CSV. Gracias a este proceso, puedes luego analizar el comportamiento de tu protocolo TDMA, revisar la evolución de las variables de los robots o depurar la lógica de comunicación usando herramientas como Excel, pandas o MATLAB.

El Flujo del código nuevo se puede ver a la derecha y el antiguo a la izquierda.