UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Informática de Sistemas y Computadores

Comunicación tolerante a fallos entre vehículos autónomos no tripulados

Trabajo Fin de Máster

Máster Universitario en Ingeniería de Computadores y Redes

AUTOR/A: Vicente García, Adrián

Tutor/a: Saiz Adalid, Luis José

Cotutor/a: Gracia Morán, Joaquín

CURSO ACADÉMICO: 2022/2023

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Agradecimientos

A mis padres y mi hermana,

por el apoyo incondicional en la finalización de este proyecto.

A mi grupo de amigos del Farming, a Cristian y a Pedro por darme apoyo moral en todo momento.

Y a Carla, por ayudarme mentalmente desde el primer día.

Resumen

Este trabajo consiste en la utilización de dos prototipos de vehículos autónomos no tripulados, ambos con un sistema de conducción basado en Arduino Uno y equipados con distintos sensores, que serán capaces de comunicarse entre sí mediante módulos Bluetooth del tipo HC-05. Mediante una plataforma de pruebas a modo de circuito, dichos vehículos se moverán uno detrás de otro y en constante movimiento. Gracias a unos sensores de distancia ultrasónicos y a los módulos de conexión inalámbrica Bluetooth, el primero de ellos transmite la información correspondiente al que le sigue para monitorizar sus respuestas. Para estudiar la tolerancia a fallos y mejorar la confiabilidad de la comunicación entre los vehículos la idea principal es el uso de códigos correctores de errores para proteger la información transmitida, y la introducción de diversos fallos mediante técnicas de inyección de fallos. Con la prueba de diversos códigos se pretende valorar y analizar la mejora obtenida en la confiabilidad, y el sobrecoste en el que se incurre, cuestión importante en sistemas empotrados como los utilizados.

Palabras clave: Bluetooth, tolerancia a fallos, códigos correctores de errores, confiabilidad, sistemas empotrados.

Resum

Aquest treball consisteix en la utilització de dos prototips de vehicles autònoms no tripulats, tots dos amb un sistema de conducció basat en Arduino Uno i equipats amb diferents sensors, que seran capaços de comunicar-se entre si mitjançant mòduls Bluetooth del tipus HC-05. Mitjançant una plataforma de proves a manera de circuit, aquests vehicles es mouran un darrere d'un altre i en constant moviment. Gràcies a uns sensors de distància ultrasònics i als mòduls de connexió sense fil Bluetooth, el primer d'ells transmet la informació corresponent al que li segueix per a monitorar les seues respostes. Per a estudiar la tolerància a fallades i millorar la confiabilitat de la comunicació entre els vehicles la idea principal és l'ús de codis correctors d'errors per a protegir la informació transmesa, i la introducció de diverses fallades mitjançant tècniques d'injecció de fallades. Amb la prova de diversos codis es pretén valorar i analitzar la millora obtinguda en la confiabilitat, i el sobrecost en el qual s'incorre, qüestió important en sistemes encastats com els utilitzats.

Paraules clau: Bluetooth, tolerància a fallades, codis correctors d'errors, confiabilitat, sistemes encastats.

Abstract

This work consists of using two prototypes of unmanned autonomous vehicles, both with a driving system based on Arduino Uno and equipped with different sensors, which will be able to communicate with each other with Bluetooth modules of the HC-05 type. Using a test platform as a circuit, these vehicles will move one after the other and in constant motion. Thanks to ultrasonic distance sensors and Bluetooth wireless connection modules, the first one transmits the corresponding information to the next one in order to monitor its responses. To study fault tolerance and improve the reliability of communication between the vehicles, the main idea is the use of error-correcting codes to protect the transmitted information, and the introduction of various faults by means of fault injection technique. By testing various codes, the aim is to assess and analyze the improvement in reliability and the cost of incurred overheads, which is important in embedded systems such as the ones used.

Keywords: Bluetooth, fault tolerance, error-correcting codes, reliability, embedded systems.

Tabla de contenidos

Índice de figuras

1. Introducción

1.1 Objetivos

A medida que los sistemas empotrados aumentan de prestaciones, se incrementa la probabilidad de sufrir una mayor cantidad de fallos y errores, tanto en los componentes como en las comunicaciones. Esto provoca que los datos almacenados en memoria, y luego transmitidos, puedan sufrir errores, los cuales en última instancia pueden provocar averías o malfuncionamientos no deseados.

Por estos motivos, los sistemas informáticos implementan métodos o estrategias de prevención ante caídas que garantizan la continuidad y la seguridad de las operaciones que pudieran afectar a uno o varios componentes, conllevando una parada intermitente de la funcionalidad que ofrecen.

En este trabajo se van a utilizar dos vehículos autónomos, ambos equipados con un sistema de conducción y seguimiento de línea de un trazado determinado basado en Arduino Uno y diferentes sensores que permiten el intercambio de datos. La idea es trabajar con un canal de comunicación asíncrono bidireccional que permitirá el intercambio de datos entre nuestros dos vehículos autónomos, etiquetados como director y seguidor, siendo este último el encargado de interpretar los datos y reproducir las instrucciones para que se comporte tal y como se programe.

En entornos críticos, tanto la memoria como el canal de comunicaciones pueden sufrir errores, por lo que es necesario uno o varios mecanismos capaces de la detección de estos para que como mínimo, el transmisor pueda ser avisado de este error y volver a enviar los datos, o manejar la corrección de estos por parte del receptor si fuera posible. El método principal de tolerancia a fallos que vamos a utilizar son los Códigos de Corrección de Errores (ECC). Estos códigos añaden bits adicionales a las cadenas de datos, denominados bits de códigos, bits de redundancia o bits de paridad, los cuales ayudan a la identificación de los errores en la trama de datos.

En este sentido, se van a programar y evaluar tres códigos distintos dedicados a la detección y corrección de errores para comprobar el funcionamiento en los vehículos autónomos. Cada uno de ellos presenta características diferentes respecto a la detección y corrección de errores, lo cual permitirá evaluar las diferencias y las respuestas del vehículo seguidor. Todos ellos permiten codificar los datos a enviar en el vehículo director y será el seguidor el encargado de decodificar ese mensaje y poder obtener los datos que permitan a dicho vehículo ejecutar ciertas instrucciones. De esta forma, se ha evaluado el funcionamiento del sistema con los siguientes códigos de corrección de errores:

- Código de Hamming extendido (SEC-DED: *Single Error Correction – Double Error Detection*).
- Código Hsiao SEC-DED.
- Código SEC-DAEC-TAED (SEC-DAEC-TAED: *Single Error Correction – Double Adjacent Error Correction – Triple Adjacent Error Detection*).
- Sistema sin tolerancia a fallos.

Para cada uno de los tres ECC se han programado dos versiones. En la primera se implementa únicamente el decodificador en ambos vehículos. De esta forma, sólo se comprueban los posibles fallos en la comunicación de datos entre emisor y receptor mediante la inserción en el software de control de una tabla o matriz de datos codificados. En este caso las codificaciones de cada dato vienen dadas y son estas las enviadas por el vehículo director. En el segundo caso, se implementa tanto el codificador como el decodificador, por lo que las codificaciones no vienen dadas y tienen que ser calculadas por cada ECC. De esta forma, se consigue comprobar la integridad de los datos en memoria además de protegerlos durante la transmisión.

Para comprobar el funcionamiento de todas las versiones, se van a inyectar los siguientes modelos de fallo:

- Fallos simples.
- Fallos dobles adyacentes.
- Fallos triples adyacentes.
- Fallos cuádruples adyacentes.

Con estos experimentos de inyección de fallos se podrá comprobar el nivel de tolerancia a fallos del vehículo esclavo, así como observar también las posibles reacciones en comparación con la no implementación de ningún mecanismo de tolerancia a fallos.

1.2 Estructura del documento

En el siguiente capítulo se describe brevemente la tolerancia a fallos y los vehículos autónomos utilizados, basados en dos proyectos previos y que sirven como base de los prototipos montados para realizar la propuesta final de este Trabajo Final de Máster.

A continuación, se explican los cambios realizados en los vehículos autónomos, añadiendo los componentes necesarios para el completo funcionamiento del sistema.

En el capítulo cuatro se detallan los diferentes Códigos de Corrección de Errores utilizados.

Después de haber visto todo el diseño y su implantación se exponen las diferentes pruebas realizadas durante el transcurso del proyecto, así como sus propias valoraciones individuales.

Como penúltimo punto se mencionan unas conclusiones técnicas, un apartado de la relación del trabajo con las asignaturas del máster, los diferentes trabajos futuros que se podrían plantear y una valoración personal.

Finalmente se enumeran una lista de documentaciones visitadas y sus fuentes, las cuales han servido como ayuda y como fuente de información.

2. Introducción a la Tolerancia a Fallos

Se puede definir la confiabilidad como la propiedad de los sistemas informáticos que permite depositar una confianza justificada en el servicio que éste proporciona, teniendo en cuenta que la vida de un sistema informático supone un cambio continuo entre el estado de funcionamiento correcto (cuando el sistema proporciona el servicio especificado) y el estado en el que el sistema está averiado (cuando el sistema no proporciona el servicio especificado).

Este cambio entre servicio especificado y no especificado está causado por lo que se denomina "Impedimentos de la confiabilidad", que se pueden definir de la siguiente forma:

- Avería: El usuario (humano u otra máquina) aprecia que el sistema no funciona bien, es decir, el servicio entregado por el sistema no es el especificado.
- Error: Es un estado incorrecto del sistema (o de uno de sus componentes). Puede ser la causa de una avería.
- Fallo: Causa u origen probable de un error. Se trata de un defecto o imperfección en el hardware o el software del sistema que al activarse provoca un error.

Esta relación es causal, tal y como se puede ver en la Figura 1.

Figura 1. Impedimentos de la confiabilidad.

Existen diferentes medios para conseguir la confiabilidad en un sistema informático:

- Prevención de fallos: Evitar que ocurran fallos.
- Eliminación de fallos: Reducir la presencia y el alcance de los fallos.
- Predicción de fallos: Estimar el número e impacto de los fallos.
- Tolerancia a los fallos: Asegurar un servicio correcto a pesar de los fallos.

Centrándonos en la tolerancia a fallos, esta es la propiedad que permite a un sistema asegurar que proporciona un servicio correcto a pesar del fallo en alguno de sus componentes. Por tanto, cuando un componente deja de operar como corresponde, este no debería influir en el correcto funcionamiento del sistema completo.

La tolerancia a fallos se consigue mediante lo que se denomina redundancia, que consiste en introducir en el sistema tiempo, información o recursos adicionales a los que se necesitan para su funcionamiento normal.

Para implementar la tolerancia a fallos en un sistema se requieren planteamientos previos que servirán para gestionar una correcta estrategia:

- ¿Qué tipo de fallos se consideran?
- ¿Cuándo debería activarse la estrategia de tolerancia a fallos?
- ¿Cómo recuperarse tras un error o cómo mitigar su efecto?
- ¿Qué componente falla?
- ¿Qué hacer con el componente fallido?

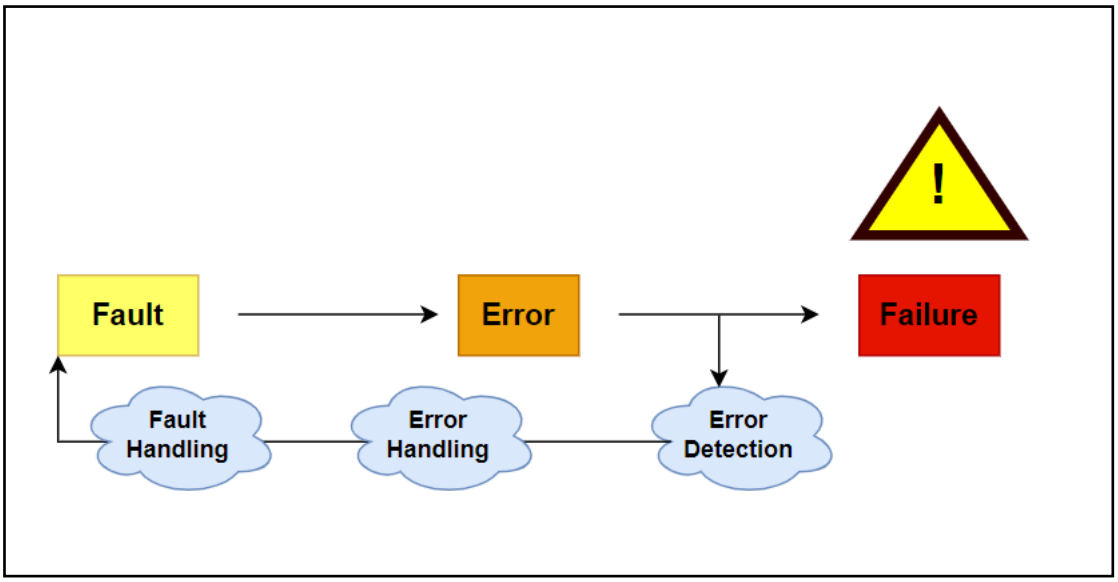

Figura 2. Propagación de fallos.

Algunos ejemplos relevantes en el término de los diseños de sistemas tolerantes a fallos pueden verse reflejados en la primera computadora que integraba esta funcionalidad, llamada SAPO y creada en la República Checa bajo la dirección del informático Antonin Svoboda. Esta tolerancia se consiguió mediante tres unidades de procesamiento que realizaban los cálculos de manera paralela. Si los resultados obtenidos diferían, se realizaba el cálculo de nuevo hasta fallar dos veces, que era cuando el sistema se detenía por completo.

3. Vehículos autónomos utilizados

Con los objetivos definidos previamente en la introducción de este documento, es necesario presentar los que serán la base principal para llegar a la solución final objetiva a mostrar en este proyecto. Por tanto, se presentan a continuación dos proyectos finales de grado en los cuales se diseña e implementan dos vehículos a escala guiados mediante el microcontrolador Arduino Uno, cada uno con características y funcionalidades específicas.

3.1 Vehículo autónomo director

En primer lugar, se debe mencionar el Trabajo Final de Grado que tiene como título "Diseño e implementación mediante aplicaciones CAD e impresión 3D de un coche a escala guiado por Arduino" cuyo autor es Jorge Lázaro Tarazón, y que fue finalizado en el curso académico 2020-2021 en la Escuela Técnica Superior de Ingeniería Industrial de Valencia.

Figura 3. Vehículo autónomo director.

El objetivo de este proyecto consistió en diseñar e implementar en primera instancia, el chasis de un vehículo autónomo para añadir posteriormente un microcontrolador Arduino del tipo Uno con una serie de sensores y actuadores que fueran capaces de simular el seguimiento de una línea durante las pruebas de recorrido en un circuito a modo de prueba para dicho vehículo.

Para la creación del vehículo propuesto se utilizaron técnicas de diseño 3D mediante aplicaciones de escritorio CAD (*Computer-Aided Design*) con las cuales se consiguió realizar los diferentes planos de las piezas requeridas que en su totalidad y con el montaje correcto harían de estos un único modelo de vehículo a escala.

Por lo que respecta al mundo de las tecnologías de la información y de la comunicación en este proyecto se eligieron los componentes necesarios para realizar la movilidad autónoma del vehículo director. El tipo de microcontrolador elegido es el Arduino Uno de la marca Elegoo, el cual servirá de base para ejecutar las órdenes grabadas en su memoria desde el Arduino IDE, además de punto central de interconexión de los sensores y actuadores en sus debidos pines. El resto de los componentes elegidos fueron:

- Motores DC: requeridos para el funcionamiento y rodamiento de las cuatro ruedas que incorpora el vehículo autónomo.

- Placa Elegoo Uno: el microcontrolador de Arduino que incluye 13 pines digitales, 5 pines analógicos, así como la entrada USB y un botón de *reset*.

- SmartCard Shield v1.0: a modo de expansión del microcontrolador, este se acopla mediante los pines e incorpora facilidad para la conexión de periféricos como motores servo, UART, sensor de línea o batería.

- Acelerómetro GY-521: este módulo incorpora un giroscopio con el que se controlan los ángulos de giro del vehículo junto a la programación correspondiente.

- Sensor ITR20001: el funcionamiento del seguimiento de línea viene controlado por este módulo. Gracias a sus tres componentes fotoeléctricos situados a izquierda, centro y derecha que responden a los cambios de intensidad lumínicos, se comprobará el funcionamiento de dicho seguimiento.

- Batería: en el mismo kit te montaje de Elegoo, se encuentra una batería de litio que se interconecta al microcontrolador para dotar a este y a sus componentes de energía cuando este está en movimiento.

- Ruedas: situadas de forma simétrica, se incorporan dos en la parte frontal y dos en la parte trasera y a las cuales se les añade un motor para proporcionarles el movimiento de giro.

- Componentes de montaje e interconexión: también son requeridos elementos de tornillería para anclar el microcontrolador al modelo del vehículo y cables para todos los sensores y actuadores que este incorpora.

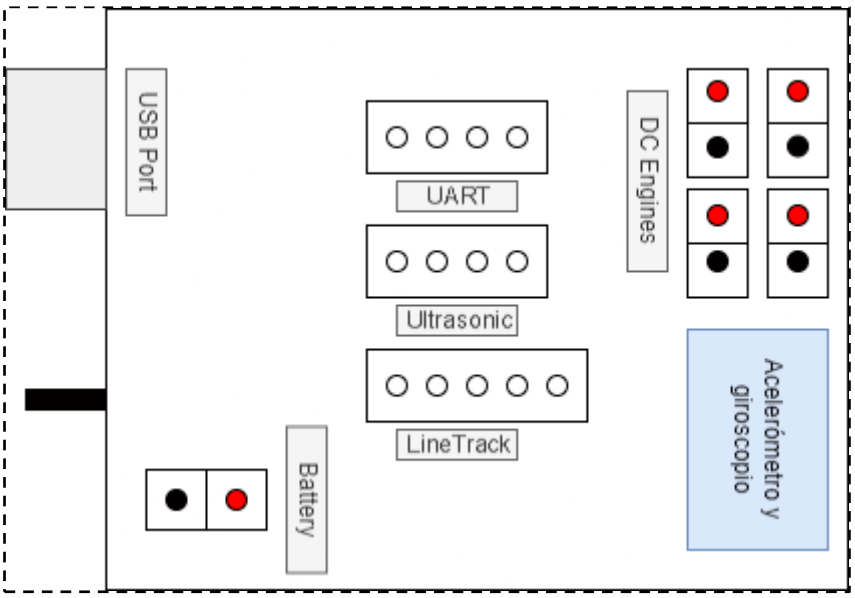

Figura 4. Esquema de componentes SmartCard Shield v1.0 vehículo director.

3.2 Vehículo autónomo seguidor

El segundo trabajo final de grado que servirá como base para la realización del presente trabajo tiene por nombre “Desarrollo de un prototipo a escala de un vehículo para pruebas de laboratorio mediante impresión 3D”, defendido en el curso académico 2020-2021 y cuyo autor es Rubén García Armero, graduado en la Escuela Técnica Superior de Ingeniería Industrial en Valencia.

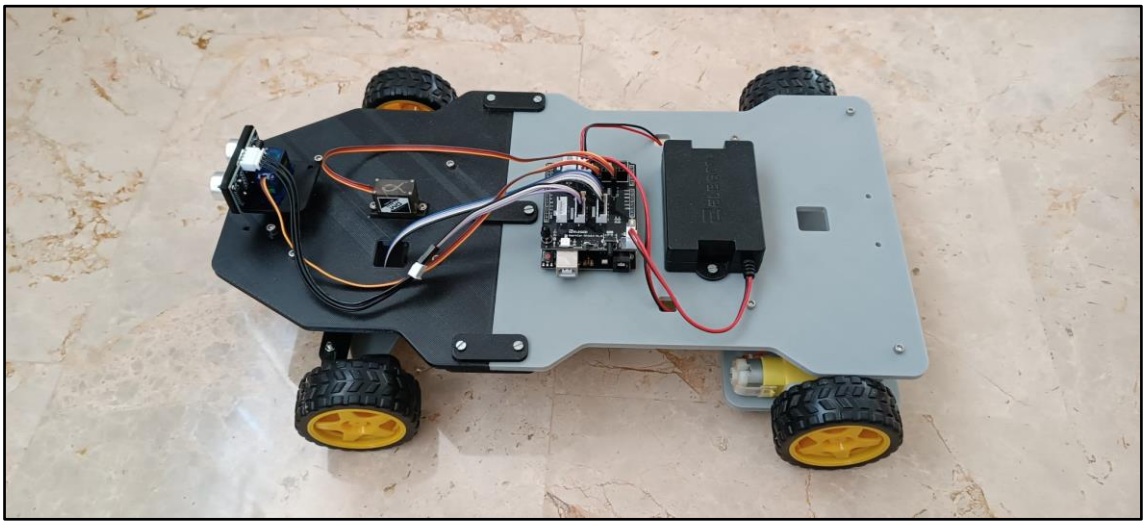

Figura 5. Vehículo autónomo seguidor.

Los dos objetivos de este trabajo fueron el diseño y la construcción de un vehículo autónomo mediante técnicas de impresión 3D y dotado de funcionalidad mediante el

microcontrolador de Arduino Uno. En este caso será necesario que se cumplan dos metas:

- Conducción libre con detección de obstáculos.

- Seguimiento de rutas detectables por el vehículo.

Para el diseño 3D de los planos requeridos se utilizó la herramienta de *Autodesk Inventor* mientras que para el modelado de las piezas se trabajó con el software *CAD Onshape* empleando técnicas de FDM (*Fusion Deposition Modeling*) las cuales permiten, mediante capas de grosor determinado, apilarse para formar piezas en tres dimensiones.

En el ámbito de la electrónica se escoge como microcontrolador el Arduino Uno, el cual aporta portabilidad y sencillez para lograr los objetivos del proyecto y que permitirá interconectar los sensores y actuadores teniendo en cuenta el número de entradas y salidas necesarias. Por lo que respecta a los elementos que conformarán la parte electrónica se escogen los siguientes:

- Placa Arduino Uno: como en el caso anterior, se elige esta placa debido a la sencillez que esta ofrece y el número de pines disponibles.

- SmartCard Shield v1.0: esta placa de expansión dispone de lo necesario para garantizar el funcionamiento de sensores y actuadores.

- Sensor ITR20001: el funcionamiento del seguimiento de línea viene controlado por este módulo. Gracias a sus tres componentes fotoeléctricos situados a izquierda, centro y derecha que responden a los cambios de intensidad lumínicos, se comprobará el funcionamiento de dicho seguimiento.

- Sensor ultrasonido HC-SR04: la forma de detección de obstáculos en este proyecto viene dada por dos sensores de ultrasonidos situados tanto en la parte delantera como en la lateral del vehículo. Este módulo emite pulsos de sonido mediante su parte emisora que rebotan volviendo a este mismo módulo, por la que utilizando la fórmula de conversión correcta se puede llegar a deducir la distancia entre el objeto encontrado y el módulo.

- Alimentación: dado el objetivo de la autonomía requerida por el vehículo, y la energía necesaria por sus componentes, también se adquiere una batería.

- Servomotor SG90: encargados de la dirección del vehículo, estos motores dotarán de radios de giro tanto al sensor ultrasonido principal como al eje de las ruedas delanteras.

- Motor de escobilla DC: en este caso y a modo de propulsión del vehículo, se emplean dos motores situados en las dos ruedas traseras.

3.3 Modificaciones realizadas

La idea principal propuesta es, a grandes rasgos, enlazar estos dos vehículos mediante comunicación inalámbrica entre ambos. Para ello, se va a necesitar de algún tipo de conexión inalámbrica como podría ser la tecnología Wi-Fi, Bluetooth o ZigBee.

La tecnología de interconexión elegida para que ambos vehículos puedan comunicarse bidireccionalmente es el Bluetooth. Para ello existen distintos módulos compatibles con las placas Arduino, aunque los escogidos en este caso serán del tipo HC-05.

Dentro de los aspectos más relevantes para la elección de esta tecnología residen en la no necesidad de ningún elemento intermediario, bastará con los dispositivos a interconectar, y la velocidad de transmisión de los datos, pues esta tecnología tiene una velocidad más que suficiente y no es necesario utilizar mucha batería en ninguno de los dos vehículos.

Las ventajas de la tecnología Bluetooth en comparación con la tecnología Zigbee son las siguientes:

- Amplia disponibilidad de dispositivos, lo que facilita la interconexión entre diferentes tipos.
- Compatibilidad con dispositivos móviles, que resulta imprescindible para algunas de las pruebas unitarias realizadas en este proyecto.
- Arquitectura de conexión punto a punto, lo que significa que dos dispositivos pueden conectarse entre sí sin la necesidad de infraestructura adicional.

Las ventajas de la tecnología Bluetooth en comparación con la tecnología Wi-Fi son las siguientes:

- Consumo de energía. La tecnología Bluetooth consume menos energía que la tecnología Wi-Fi por lo que es ideal para alimentar baterías las cuales dispondrán de una duración más prolongada.
- La simplicidad de configuración permite un establecimiento de la conexión entre dispositivos de manera menos compleja.
- Menor interferencia debido a la banda de frecuencia en la que Bluetooth trabaja. Las redes Wi-Fi cercanas podrían suponer congestión y afectar al rendimiento de la conexión.

3.3.1 Componentes del coche director

El coche director va a ser el encargado de trasmitir la información asociada a las instrucciones que deberán ser ejecutadas por el vehículo seguidor. Para ello, y partiendo de la base de los prototipos explicados anteriormente, es necesario contemplar los cambios realizados, de qué manera se interconectan todos los componentes y cómo se configuran.

La placa *SmartCar-Shield-v1.0* montada sobre el microcontrolador Arduino en el primer prototipo permite la interconexión de todos los componentes, tal y como se puede ver en la figura nº4.

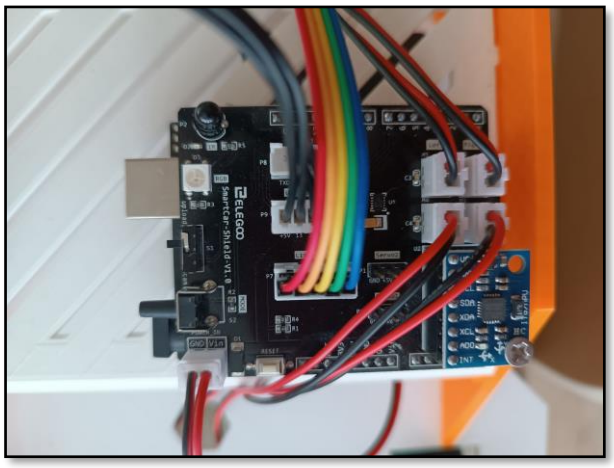

Figura 6. Placa Arduino vehículo director.

Las cuatro ruedas del vehículo incorporan, mediante un acople específico, cuatro motores DC los cuales se conectan a la placa mediante dos cables de alimentación y toma de tierra a los pines *M1* y *M4* para los motores de las ruedas de la parte izquierda y *M2* y *M3* para los motores de la parte derecha.

La batería recargable también cuenta con dos cables de voltaje y toma tierra asociados con los pines *GND* y *Vin* respectivamente situados en la etiqueta de la placa de *POWER_IN*.

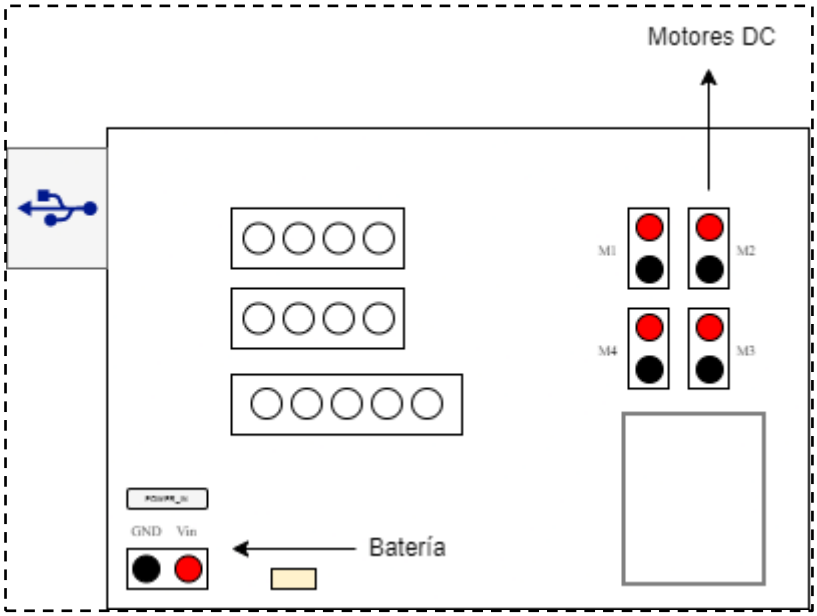

Figura 7. Esquema de conexiones vehículo director (1).

El módulo GY-521 se acopla a los pines marcados como MPU6050 en la placa y proporciona las funciones de acelerómetro y giroscopio las cuales serán necesarias para la dirección del vehículo en función del sensor de seguimiento de línea ITR20001.

El módulo ITR20001 cuenta con 5 conexiones a modo de cables que se interconectan con el slot de cinco pines de la placa etiquetado como *LineTrack*. Estos pines son: *GND*, para la toma de tierra; *+5V*, para la alimentación y los pines *A2*, *A1* y *A0* los cuales conectan los fotosensores izquierdo, medio y derecho respectivamente.

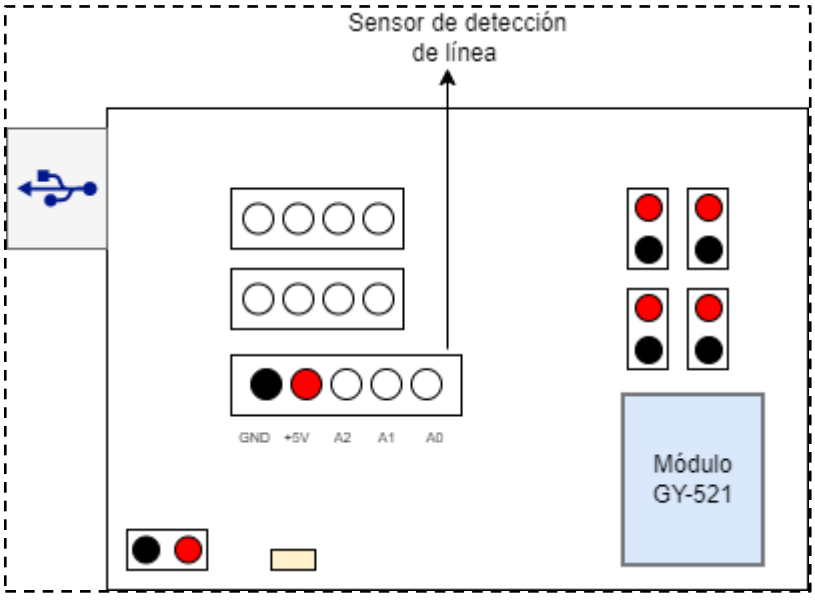

Figura 8. Esquema de conexiones vehículo director (2).

Todos los componentes o módulos mencionados previamente no han necesitado una configuración de parámetros específica ya que la funcionalidad que traían era la adecuada para los objetivos de este proyecto.

El último componente es el módulo Bluetooth HC-05. La conectividad de este a la placa viene definida por seis pines que se conectan por cable de los cuales únicamente se utilizan cuatro. Estos son los ya mencionados previamente y necesarios para cualquier componente eléctrico, *GND* como referencia a tierra y *VCC* para el suministro de energía además de los pines de transmisión y recepción de datos mediante el módulo: *TxD* y *RxD*. En la placa se encuentra mediante la etiqueta *UART* los cuatro pines a los cuales se deben conectar los cables del módulo.

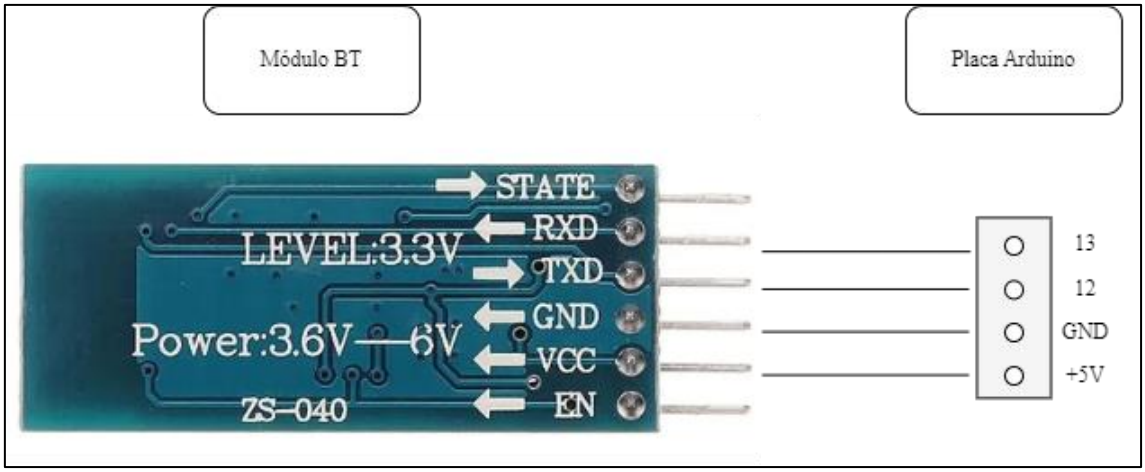

Figura 9. Conexión pines HC-05 y Arduino.

En este caso es necesario tener en cuenta una asignación de pines de la placa con los pines de transmisión y recepción de datos del módulo. En este, el pin *RxD* del módulo conecta al pin 13 de la placa Arduino mientras que el pin *TxD* del módulo conecta al pin 12 de la placa.

- Trasmisión: para la transmisión de datos es necesario conectar correctamente el pin *TxD* del módulo HC-05 al pin 12, el cual se debe configurar como pin de recepción de datos. Este pin de transmisión está destinado a trasmitir datos desde el módulo hasta la placa, y es esta la que procesa y utiliza estos datos.

- Recepción: para la recepción de datos es necesario conectar correctamente el pin *RxD* del módulo HC-05 al pin 13, el cual se debe configurar como pin de trasmisión de datos. Este pin de recepción permite la llegada de datos desde la placa al módulo y a su vez puedan ser enviados a través de la conexión Bluetooth.

Todas estas conexiones son estrictamente necesarias para que la comunicación entre los módulos HC-05 del vehículo seguidor y del vehículo director sea bidireccional y asíncrona.

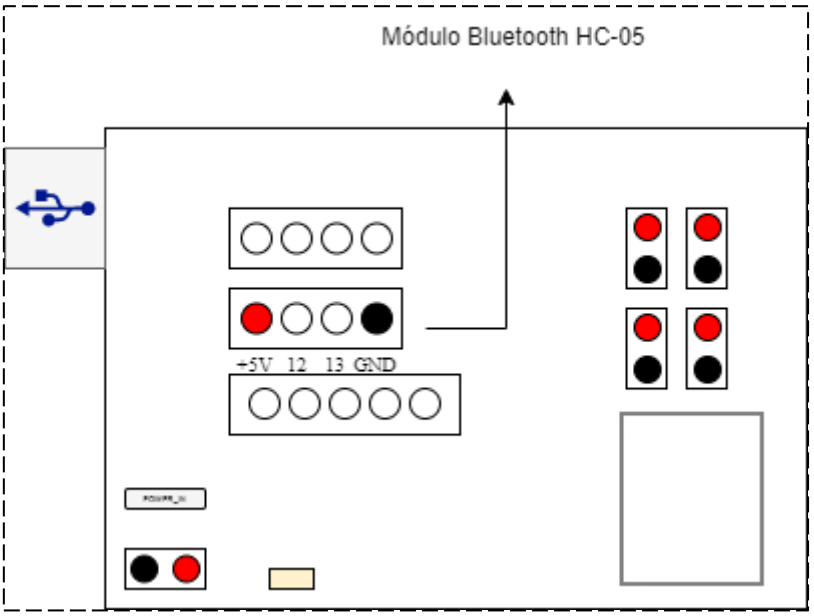

Figura 10. Esquema de conexiones vehículo director (3).

Una vez descrito como el módulo debe conectarse a la placa para obtener el funcionamiento deseado, se define ahora como este módulo debe ser configurado internamente para que realice las tareas descritas en los objetivos de esta memoria.

El coche director es el encargado de enviar los datos codificados mediante los códigos propuestos para que el vehículo seguidor realice las instrucciones programadas internamente en su código. Para ello se crea una canal de conexión inalámbrica mediante los módulos Bluetooth de manera que los datos puedan ser transmitidos y recibidos en ambos lados de la comunicación.

El módulo Bluetooth conectado en el vehículo director debe estar configurado de tal manera que desempeñe un rol de maestro. De esta forma será el encargado de iniciar, controlar y establecer la conexión con el dispositivo Bluetooth esclavo del coche seguidor. Se configura mediante el software de Arduino IDE los siguientes parámetros mediante los comandos AT introducidos en el monitor serie:

Comando	Valor
AT + ROLE	1
AT + UART	38400,0,0
AT + VERSION	
AT + NAME	CocheDir
AT + CMODE	0
AT + BIND	18:E4:400006
AT + RESET	

Figura 11. Comandos AT vehículo director.

- AT +ROLE = 1→ con este comando se le indica al módulo que asuma el rol de maestro para que inicie la conexión Bluetooth.

- AT + UART = 38400,0,0→ la velocidad de transmisión del UART debe estar configurada de tal manera que sea posible introducir los comandos AT. Una vez configurado de esta forma puede inicializarse a 9600 baudios también, aunque en este proyecto se establece a la velocidad mencionada.

- AT+VERSION→ este comando permite obtener los datos referidos a la versión del *firmware* del módulo. De esta forma se comprueba su compatibilidad con los comandos y con la versión de *firmware* del módulo configurado en el vehículo seguidor.

- AT + NAME = CocheDir → para las pruebas unitarias realizas con un teléfono móvil particular es necesario establecer un nombre para la detección Bluetooth por parte de este. Además, permite asignar un nombre a modo de identificación.

- AT + CMODE = 0 → con el valor 0 se indica al módulo que deberá conectarse a un dispositivo particular y no a cualquier dirección MAC disponible.

- AT + BIND = 18: E4: 400006 → esta es la dirección MAC del módulo HC-05 conectado en el vehículo seguidor a la que el módulo maestro debe conectarse. Este comando está relacionado con AT + CMODE.

- AT + RESET → cuando la configuración está terminada es necesario reiniciar el módulo para que los cambios queden guardados y estén activados en el próximo encendido.

La configuración del módulo está completa y se debe ahora realizar la del módulo seguidor. Pero antes se debe especificar en el software Arduino IDE donde se programa el código de qué manera debe conectarse, así como distintos parámetros de entrada para que la conexión por el canal de comunicación se realice como se espera.

```
#include <avr/wdt.h>
#include "DeviceDriverSet_xxx0.h"
#include "ApplicationFunctionSet_xxx0.h"
#include <SoftwareSerial.h> // Incluimos la librería SoftwareSerial
SoftwareSerial BTserial(12, 13); //RX y TX
byte myByte1;
unsigned long tiempoAnterior = 0;
```

Figura 12. Inicialización del puerto Bluetooth.

En el código del vehículo director se añade la librería *SoftwareSerial.h* la cual permite la comunicación serial en pines digitales de componentes que no son nativos de Arduino. Gracias a esta se añaden funcionalidades necesarias como el establecimiento de un canal de comunicación virtual, envío y recepción de datos, así como configurar la velocidad de transmisión. Se incluye en el código y permite añadir métodos relacionados con la conexión Bluetooth HC-05.

En la segunda línea se crea un objeto de la clase *SoftwareSerial* y es utilizado para instanciar dicho objeto de comunicación serial donde el número 12 representa el pin digital utilizado para recibir datos y el número 13 para enviar datos. De esta forma ya se podría establecer una comunicación serial con otros dispositivos que también estén correctamente configurados.

```
void setup()
{
  Application_FunctionSet.ApplicationFunctionSet_Init();
  BTserial.begin(38400); // Iniciamos comunicación serial con el módulo HC-05
  Serial.begin(38400); // Inicializamos comunicación con puerto serie de Arduino IDE
  myByte1 = 0b0001;

}
```

Figura 13. Inicialización de la comunicación.

Una vez importada la librería correspondiente y la configuración de los pines de recepción y envío de datos es necesario iniciar la comunicación. En el método *setup* predefinido por Arduino IDE se establece la velocidad de transmisión de datos en baudios siendo *BTserial* el objeto creado previamente. Esta velocidad debe coincidir con la configurada mediante el comando AT+UART.

En el caso de *Serial.begin* se establece la velocidad de transmisión entre la placa Arduino y el dispositivo conectado a través del puerto serie, es decir, el ordenador personal en el que se desarrolla el proyecto. Se ha inicializado también a 38400 baudios y debe coincidir con la velocidad de transmisión del monitor serie de Arduino IDE, además de seleccionar la opción de Ambos NL & CR la cual indica el final de una línea de texto tras el envío o la recepción de un dato.

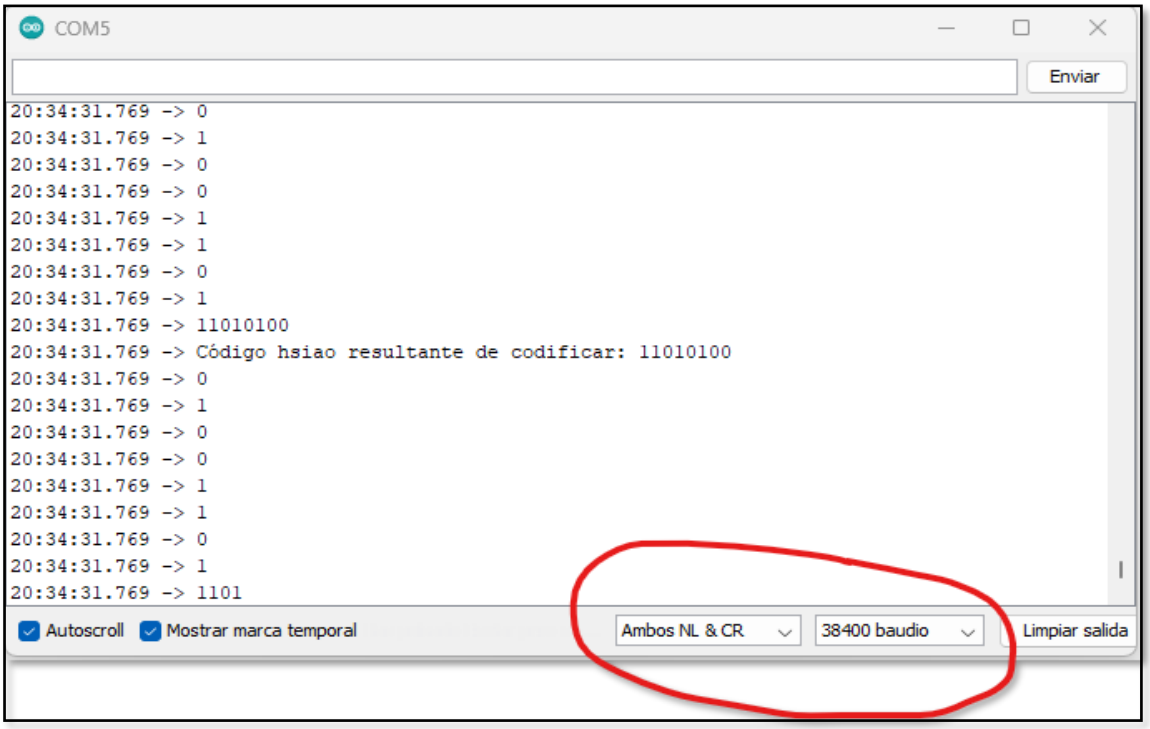

Figura 14. Configuración del monitor serie.

Tras las conexiones y configuraciones detalladas previamente el vehículo director ya dispone de lo necesario para comenzar una comunicación Bluetooth con el vehículo seguidor, en cuanto este esté configurado también. Asimismo, se pueden implementar los códigos ECCs descritos y comprobar la tolerancia a fallos de los prototipos.

3.3.2 Componentes del coche seguidor

El coche seguidor será el encargado de recibir los datos codificados por el canal de comunicación Bluetooth y será este quien los decodifique correctamente según el código implementado para que pueda ejecutar las instrucciones asociadas a cada dato. Como se mencionaba con el coche director, y partiendo de la base sobre los prototipos entregados, se debe estudiar y analizar de qué manera se interconectan todos los componentes y como se configuran para llegar a obtener el objetivo final.

Al igual que ocurre con el vehículo director, este también lleva montado la misma *SmartCar-Shield-v1.0* sobre el microcontrolador Arduino que incorpora el vehículo director y con el cual se interconectan los componentes necesarios mediante los pines.

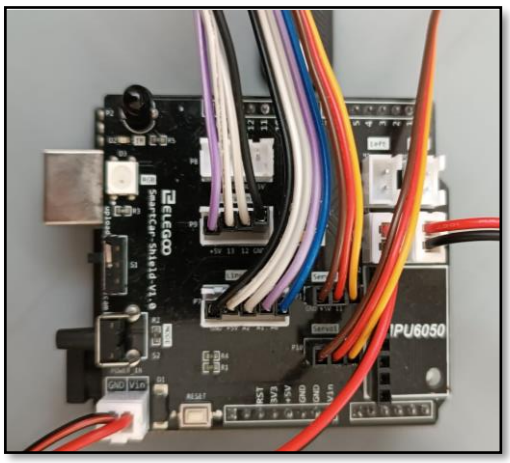

Figura 15. Placa Arduino vehículo seguidor.

Mediante las cuatro ruedas del vehículo se permite el movimiento por el circuito propuesto, pero en este caso se incorporan también mediante un acople específico únicamente dos motores de corriente continua (DC) los cuales se conectan a la placa mediante los cables de alimentación y toma de tierra al pin *M4* para el funcionamiento del motor trasero izquierdo mientras que para la rueda de la parte derecha trasera se utiliza el pin *M3*. Por tanto, la potencia de los motores se proporciona a las ruedas traseras y las ruedas delanteras se encargan de la dirección del vehículo.

La batería recargable suministra la energía requerida en todo momento por los componentes montados. El cable de voltaje se conecta al pin *Vin* mientras que el cable para la toma de tierra se conecta al pin *GND*, ambos sobre la etiqueta de *POWER_IN*.

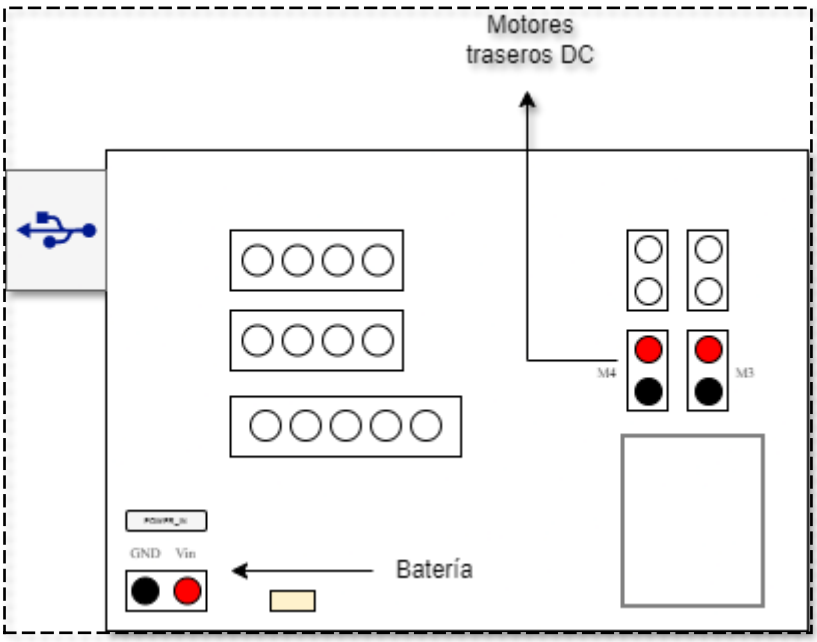

Figura 16. Esquema de conexiones vehículo seguidor (1).

Para la dirección del vehículo no se requiere de un módulo con giroscopio y acelerómetro. En este caso se incorpora un servomotor de corriente continua para controlar la posición y establecer el movimiento deseado del vehículo según la programación de su código. De esta manera se enlaza el funcionamiento del módulo de seguimiento de línea con el del servomotor para que realice la trazada del circuito en todo momento. Se necesita conectar a la placa a los pines de alimentación y toma de tierra, además del pin 10 que controla la funcionalidad del servomotor en el *slot* etiquetado como *Servo1*.

El seguimiento de línea se produce gracias al módulo ITR20001 el cual cuenta con cinco conexiones a modo de cables que se interconectan con el *slot* de cinco pines de la placa etiquetado como *LineTrack*. Estos pines son: *GND*, para la toma de tierra; *+5V*, para la alimentación y los pines *A2*, *A1* y *A0* los cuales conectan los fotosensores izquierdo, medio y derecho respectivamente.

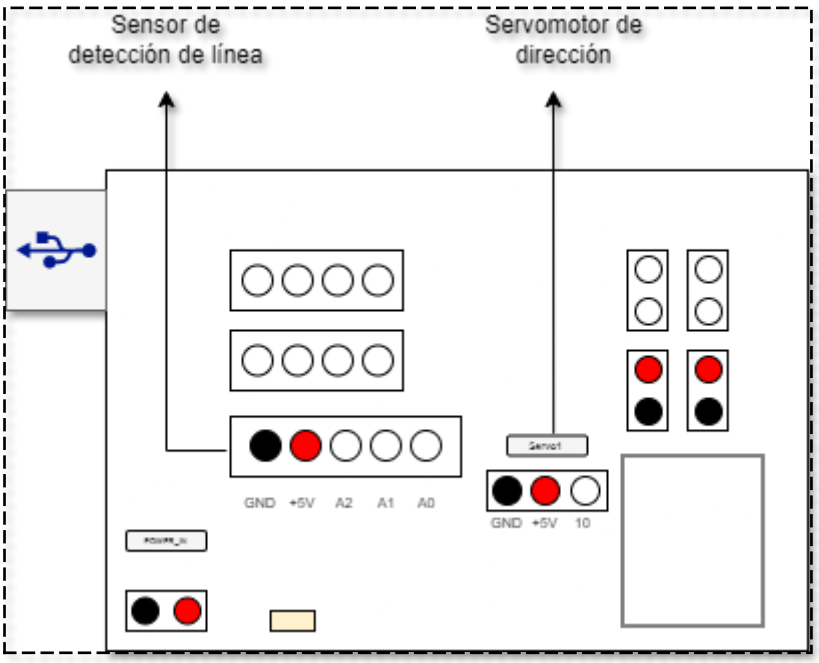

Figura 17. Esquema de conexiones vehículo seguidor (2).

El último componente que se incorpora y es imprescindible para lograr los objetivos del proyecto es el módulo Bluetooth HC-05. La conectividad de este a la placa viene definida en el capítulo anterior en el que se explican los pines y como conectarlos, ya que en este caso es exactamente igual debido al uso de la misma placa en ambos vehículos.

Por supuesto, todas esas conexiones son estrictamente necesarias para que la comunicación entre los módulos HC-05 de este vehículo y del director sea bidireccional y asíncrona.

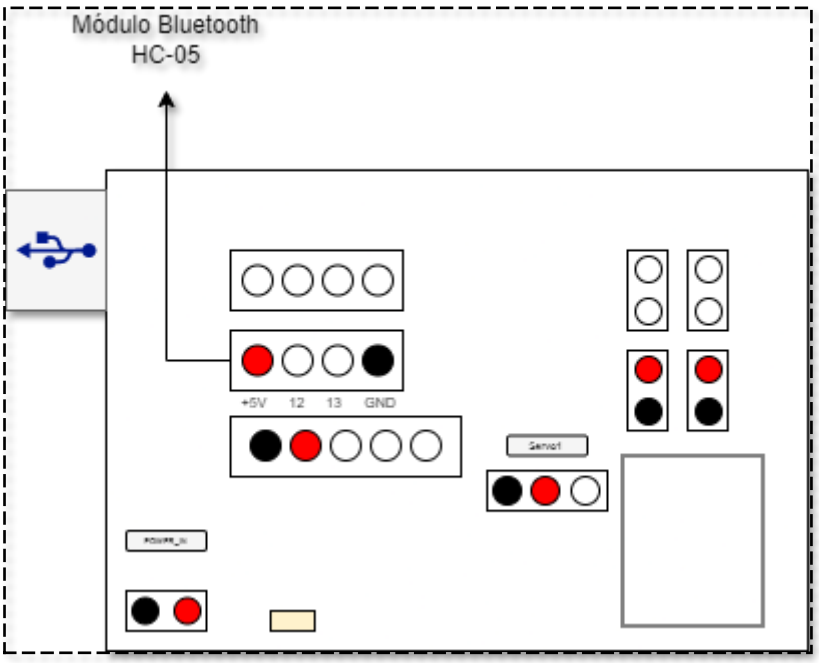

Figura 18. Esquema de conexiones vehículo seguidor (3).

El esquema de los componentes interconectados está descrito completamente y es necesario definir como el módulo Bluetooth debe comportarse internamente en el vehículo seguidor.

El vehículo seguidor es el encargado de recibir los datos codificados mediante los códigos propuestos y por el canal de comunicación inalámbrico Bluetooth. Una vez recibidos, internamente esos datos se decodifican y se sacan las instrucciones que representan cada dato.

La manera de configurar el módulo HC-05 en el vehículo seguidor es mediante la asignación del rol de esclavo, principalmente. Tiene la función de estar preparado para responder a la solicitud de conexión del módulo maestro y una vez aceptada se establece la conexión y su canal de comunicación para la transmisión de datos. Se utilizan varios comandos AT para configurar la funcionalidad requerida en este vehículo mediante el software de Arduino IDE para configurar mediante el software de Arduino IDE los siguientes parámetros mediante los comandos AT introducidos en el monitor serie.

Comando	Valor
AT + ROLE	0
AT + UART	38400,0,0
AT + VERSION	
AT + NAME	CocheSeg
AT + ADDR	
AT + RESET	

Figura 19. Comandos AT vehículo seguidor.

Los comandos introducidos pueden requerir de la escritura de un valor o no. En el caso de alguno de ellos es necesario introducir un valor que especifique como va a funcionar el módulo. El resto son informativos y muestran datos requeridos para la compatibilidad y la conexión entre ambos dispositivos.

- AT +ROLE = 0→ con este comando se le indica al módulo que asuma el rol de esclavo para que se conecte a una conexión Bluetooth.

- AT + UART = 38400,0,0→ la velocidad de transmisión del UART debe estar configurada de tal manera que sea posible introducir los comandos AT. Una vez configurado de esta forma puede inicializarse a 9600 baudios también, aunque en este proyecto se establece a la velocidad mencionada.

- AT+VERSION→ se introduce este comando para averiguar la versión de firmware del dispositivo y comprobar la compatibilidad con el módulo maestro. La versión mostrada es la 3.0-20170601 que indica la versión principal y la fecha de generación del *firmware*.

- AT + NAME = CocheSeg → se define también un nombre identificador del módulo Bluetooth para las pruebas unitarias de conexión con un teléfono móvil particular.

- AT + ADDR → la dirección MAC de cada módulo se puede obtener introduciendo este comando. Para el comando AT+BIND del módulo maestro se requiere de la MAC del módulo esclavo para poder conectarse

- AT + RESET → cuando la configuración está terminada es necesario reiniciar el módulo para que los cambios queden guardados y estén activados en el próximo encendido.

Con estos comandos se finaliza la configuración interna del módulo esclavo. Ahora se debe especificar en el software Arduino IDE donde se programa el código del vehículo seguidor al igual que se hace con el vehículo director.

```
#include <SoftwareSerial.h> // Incluimos la librería SoftwareSerial
SoftwareSerial BTserial(12, 13); // Definimos los pines RX y TX del Arduino conectados al Blue
Servo servo2;
int estado, estadoanterior;
```

Figura 20. Inicialización puerto Bluetooth.

Se importa también la librería *SofwareSerial.h* y se crea un objeto se su mismo tipo llamado *BTSerial* con los pines de transmisión y recepción definidos. Se establecen en el método *setup* y mediante el método *begin* y la velocidad de transmisión entre placa Arduino y ordenador personal siendo ambos de 38400 baudios.

Una vez importada la librería correspondiente y la configuración de los pines de recepción y envío de datos es necesario iniciar la comunicación. En el método *setup* predefinido por Arduino IDE se establece la velocidad de transmisión de datos en baudios siendo *BTserial* el objeto creado previamente. Esta velocidad debe coincidir con la configurada mediante el comando AT+UART.

En el caso de *Serial.begin* se establece la velocidad de transmisión entre la placa Arduino y el dispositivo conectado a través del puerto serie, es decir, el ordenador personal en el que se desarrolla el proyecto. Se ha inicializado también a 38400 baudios y debe coincidir con la velocidad de transmisión del monitor serie de Arduino IDE, además de seleccionar la opción de "Ambos NL & CR" la cual indica el final de una línea de texto tras el envío o la recepción de un dato.

3.4 Tecnologías y componentes hardware

Como se ha comentado durante la memoria de este proyecto, han sido muchas las tecnologías y herramientas software y hardware requeridas para llegar a los objetivos finales. A continuación, se describen cada uno de ellos detalladamente.

- Microcontrolador Arduino Uno

Arduino es una plataforma electrónica de código abierto que ofrece una combinación de hardware y software flexibles y de fácil uso para los creadores y desarrolladores. Los microordenadores de una sola placa que se pueden crear con esta plataforma tienen múltiples aplicaciones que la comunidad de creadores puede explorar.

Este proyecto fue diseñado en el año 2003 como solución a los altos costes que los estudiantes italianos tenían con los microcontroladores BASIC Stamp. Con la ayuda de los cinco cofundadores de Arduino, el proyecto no ha dejado de expandirse, así como de integrar cada vez más a más usuarios a la comunidad.

Cuando hablamos del software nos referimos al entorno de desarrollo integrado de Arduino, que nos proporcionará las herramientas necesarias para el desarrollo de código como programadores. El lenguaje está basado en C++ y los programas reciben el nombre de *sketch*. Además, es multiplataforma y se puede descargar fácilmente desde la propia web del proyecto (Arduino IDE).

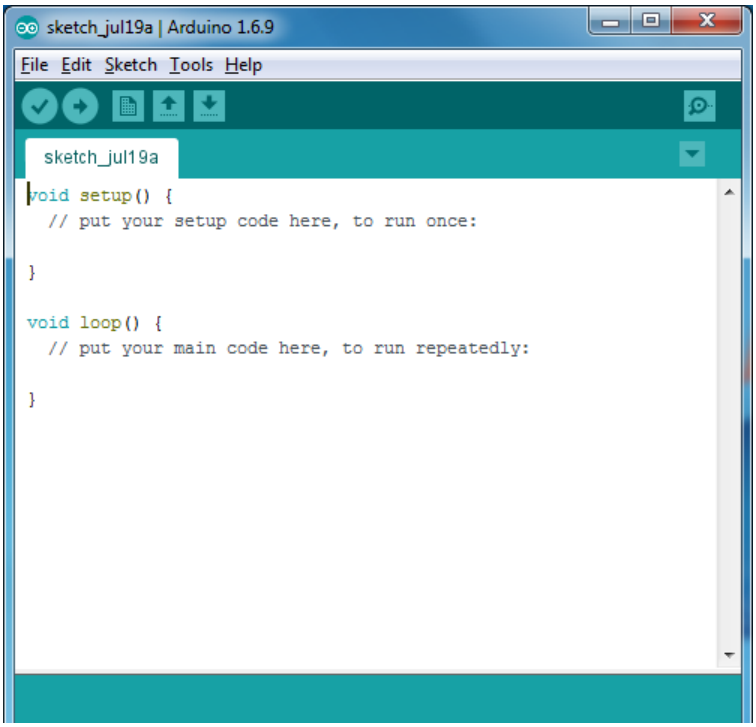

Figura 21. Sketch Arduino IDE.

La parte del hardware se refiere la placa impresa que incluye al microcontrolador de tipo Atmel y a los diferentes pines (analógicos, digitales, alimentación y toma de tierra) y a los conectores. Para hacerla funcionar basta con alimentar la placa mediante un cable USB conectado al ordenador desde donde se le vuelca el código.

La placa escogida para este proyecto es la del tipo Arduino Uno, utilizada e integrada en ambos modelos de vehículos. La placa Arduino Uno es una de las más populares y utilizadas dentro de la familia de placas Arduino. Esta placa cuenta con un microcontrolador ATmega328P de la empresa Atmel, que es el encargado de controlar y ejecutar las instrucciones que se le envían a través del código que se programa en ella.

La placa Arduino Uno tiene un total de 14 pines digitales de entrada/salida, de los cuales 6 pueden ser utilizados como salidas PWM (modulación por ancho de pulso) y 6 como entradas analógicas. Además, cuenta con un puerto USB para conectarla a un ordenador y poder programarla, y un conector de alimentación para conectarla a una fuente de energía externa.

Otra característica importante de la placa Arduino Uno es que tiene un cristal oscilador de 16 MHz, lo que le permite tener una precisión temporal adecuada para la mayoría de las aplicaciones. También tiene un botón de *reset* que se puede utilizar para reiniciar el programa que se está ejecutando en la placa. Además, incluye una memoria SRAM de 2KB, una memoria *flash* de 32KB y una memoria EEPROM de 1KB.

En cuanto a la programación de la placa, se utiliza el lenguaje de programación Arduino, que es una versión simplificada de C++. La programación se realiza a través de un entorno de desarrollo integrado (IDE) que se puede descargar gratuitamente desde la página web de Arduino. Una vez que se ha escrito el código, se puede cargar en la placa a través del puerto USB.

En resumen, la placa Arduino Uno es una herramienta electrónica de código abierto y fácil de usar que permite crear una gran variedad de proyectos y aplicaciones. Con sus numerosos pines de entrada/salida, puertos de comunicación y su fácil programación, es una excelente opción para principiantes y expertos por igual.

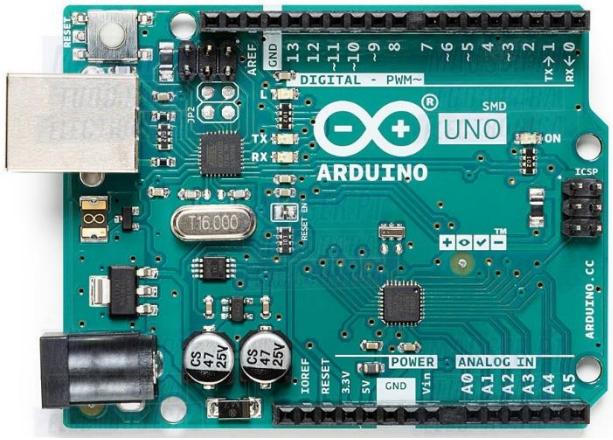

Figura 22. Microcontrolador Arduino IDE.

- Modulo Bluetooth HC-05

Para la comunicación entre ambos vehículos, se ha utilizado la tecnología Bluetooth. Concretamente se han incorporado dos módulos de protocolo de puerto serie del tipo HC-05, siendo el modo de comunicación de tipo UART. El significado de este tipo de conexión es *Universal Asynchronous Receiver/Transmitter* el cual es un protocolo simple de dos hilos para el intercambio de datos en serie. En resumen, al ser asíncrono no hay reloj compartido y en ambos lados de la conexión debe configurarse la misma velocidad de transmisión.

El módulo HC-05 es un módulo de comunicación inalámbrico que se puede utilizar en placas como las de la marca Arduino para permitir el envío y la recepción de datos desde un dispositivo a otro utilizando una interfaz serie. Esto quiere decir que mediante el software de Arduino IDE se pueden utilizar las funciones serie para enviar y recibir mensajes.

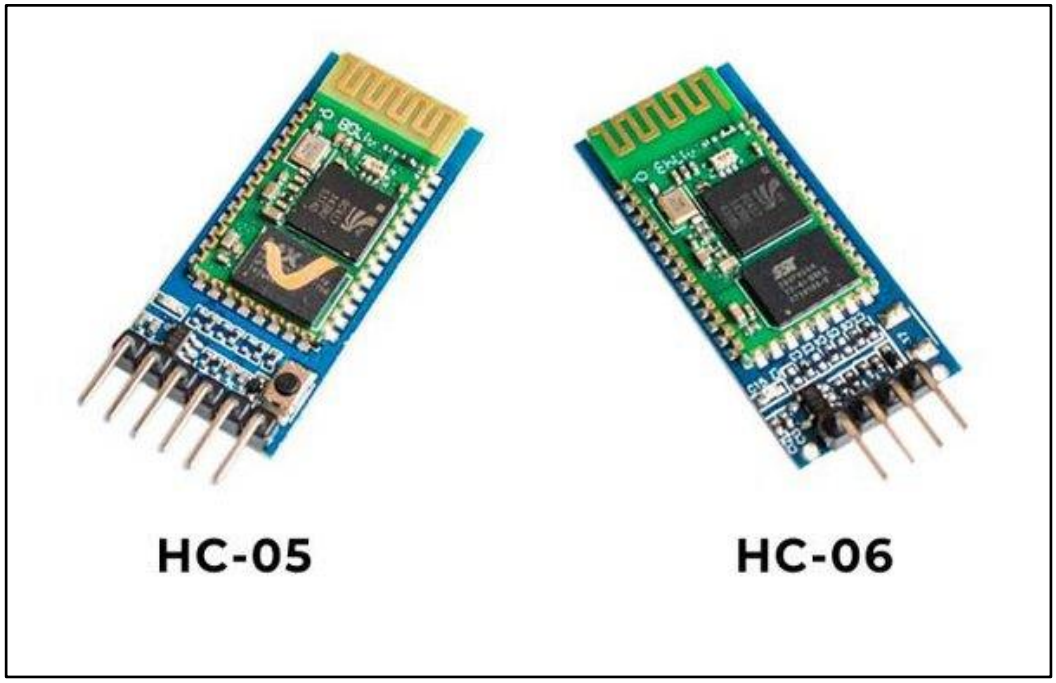

Figura 23. Módulos HC-05 y HC-06.

Dentro de los motivos para la elección de este módulo se encuentra la capacidad que este tiene para ser configurado como esclavo o como maestro. Por defecto, vienen con el modo esclavo, pero haciendo uso de los comandos AT se puede reconfigurar uno de ellos para trabajar en modo maestro. Además, dentro de las especificaciones técnicas que ofrece, encontramos un alcance máximo de 10 metros, unas dimensiones de aproximadamente 37*16 mm que nos permite conectarlo en cualquier lugar de nuestro modelo de vehículo o una velocidad de transmisión adecuada entre los 1200 bps hasta 1,3 Mbps.

La manera de conectarlo a la placa Arduino Uno es mediante los diferentes pines que este tiene:

- VCC: Este es el pin utilizado para proporcionar la alimentación requerida por el módulo y debe conectarse al pin de 5V de la placa.

- GND: El pin de toma de tierra que debe conectarse también con el pin *GND* de la placa.

- TXD: Requerido para la transmisión de datos hacia otro dispositivo y debe conectarse con el pin *RxD* del microcontrolador.

- RXD: Requerido para la recepción de datos desde otro dispositivo y debe conectarse con el pin *TxD* del microcontrolador.

- EN: Este pin es la entrada de habilitación y se utiliza para cambiar el modo del módulo entre el modo de comandos AT y el modo de comunicación inalámbrica. Se conecta a cualquier pin de salida digital de la placa Arduino, como el pin 2. No siempre se requiere conectarlo.

- STATE: Este pin es la salida de estado y se utiliza para indicar el estado actual del módulo, como si está conectado o no, o si está en modo de comandos AT o en modo de comunicación inalámbrica.

Como se mencionaba previamente, este módulo (a diferencia de otros como el HC-06) puede ser configurado tanto como esclavo como maestro. Si el módulo de Bluetooth está configurado como esclavo, está preparado para escuchar peticiones de conexión y así recibir datos. En este modo no pueden iniciar ninguna conexión, mientras que en el modo maestro es este quien envía datos e inicia y establece la conexión con un módulo en modo esclavo.

El HC-05 viene de fábrica preconfigurado como esclavo y para establecerlo en modo maestro se necesita entrar en el modo de comandos AT y configurar los parámetros para que al menos un módulo funcione en modo maestro. Para iniciar este modo es requerido que a la hora de conectar el módulo a la alimentación mantengamos pulsado el botón de *reset* del HC-05. Si se ha iniciado en modo AT, se aprecia como el parpadeo de este es más lento que en el modo conexión. Además, dentro del monitor serie de Arduino IDE, si se envía el comando AT debemos recibir un “OK” por pantalla. El mismo resultado se obtiene si se determina el resto de los valores correctamente. Por lo tanto, se requiere configurar los siguientes parámetros como mínimo para configurar un módulo como maestro:

- AT+ROLE → Con este comando se indica al módulo el modo en el que tiene que funcionar, siendo el 0 esclavo y el 1 maestro.

- AT+NAME → Para proporcionar un nombre al módulo.

- AT+CMODE → Elige entre 0 si se requiere conectar a una dirección específica o 1 si se necesita conectar a una dirección aleatoria.

- AT+BIND → Con este comando se especifica la dirección a conectar. En este caso se debe pasar como valor la dirección MAC del módulo esclavo. Se envía de la siguiente forma: 1234,56,ABCDEF, la cual equivale a la dirección 12:34:56:AB:CD:EF.

- AT+RESET → Una vez creada la configuración, es necesario mandar este comando para que se aplique y vuelva al modo de conexión, saliendo del modo AT.

Una vez configurado cada uno de los módulos como esclavo y maestro, estos se sincronizarán y conectarán, lo que permitirá el envío y recepción de datos mediante el código programado en el Arduino IDE.

- Sensor ITR20001

El sensor ITR20001 es una solución compacta y económica para la detección de obstáculos y objetos en aplicaciones de robótica, automatización y seguridad. En este caso, este sensor se monta en la parte inferior de los modelos para la detección de líneas en el suelo mientras los vehículos están en movimiento. Su tamaño reducido y su bajo consumo de energía lo hacen ideal para aplicaciones en las que el espacio y la energía son limitados. Además, su diseño de paquete único hace que sea fácil de montar en una variedad de configuraciones.

La idea principal de este componente es comportarse como sensor óptico reflectivo y para ello dispone de tres fotosensores cada uno con su propio emisor y receptor los cuales dividen la zona de detección para el monitoreo de una superficie dada.

El ITR20001 también tiene una respuesta rápida y una alta precisión en la detección de objetos. El sensor puede detectar objetos a distancias de hasta 15 mm y tiene una respuesta de frecuencia de hasta 10 kHz. Esto lo hace ideal para aplicaciones que requieren una alta velocidad de detección y una rápida retroalimentación.

Es importante tener en cuenta que el ITR20001 es un sensor analógico y requiere un circuito adicional para procesar la señal de salida del sensor. Para obtener una salida digital, se puede utilizar un circuito comparador para convertir la señal analógica en una señal digital. También puede ser necesario ajustar la distancia de detección mediante la calibración del circuito y la ubicación del sensor.

En resumen, el ITR20001 es un sensor infrarrojo de reflexión compacto y económico que se utiliza comúnmente en aplicaciones de detección de obstáculos y posicionamiento de objetos. Su capacidad para detectar objetos independientemente del color o material del objeto, su respuesta rápida y su alta precisión lo hacen ideal para una variedad de aplicaciones en robótica, automatización y seguridad.

Respecto a los pines que este tiene, vemos que son cinco, de los cuales tres pertenecen a los fotosensores que detectan el área de la izquierda (L), de la parte media (M) y de la parte derecha (R). Y, por último, dos pines de alimentación (VCC) y toma de tierra (GND), respectivamente.

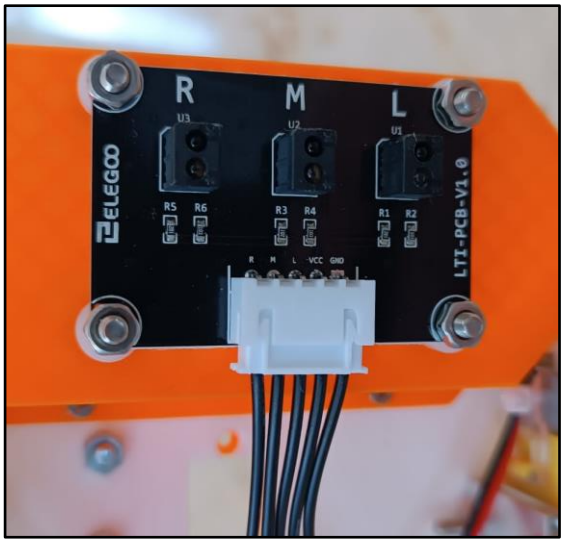

Figura 24. Sensor de detección de línea ITR20001.

4. Códigos de Corrección de Errores

4.1 Introducción a los Códigos de detección y corrección de errores (ECC)

Los códigos de Detección y Corrección de Errores permiten reducir los efectos de diferentes tipos de fallos que pueden producirse tanto en la transmisión de información como en su almacenamiento. La posibilidad de que se produzcan errores provoca que en determinados entornos se deban proteger los datos. Es necesario añadir información a esos datos para poder detectar, enmascarar y corregir errores en los mismos. Dicha acción es denominada Redundancia en la Información. Esta información extra debe seguir unas reglas para poder ser utilizada en la detección o corrección de errores.

Así pues, un código es un medio para representar datos usando un conjunto de reglas bien definidas, mientras que una palabra codificada es un elemento del código que satisface las reglas de codificación establecidas. La codificación de un dato produce una palabra codificada, y la decodificación de una palabra codificada genera un dato, tal y como se puede ver en la Figura 25. Cuando el transmisor genera un dato, este se codifica antes de enviarlo por el canal de transmisión. En este canal, puede haber ruido, lo que provoca que la palabra codificada pueda ser modificada, por lo que, al decodificarla, los datos que reciba el receptor no serán los mismos que los datos enviados por el transmisor.

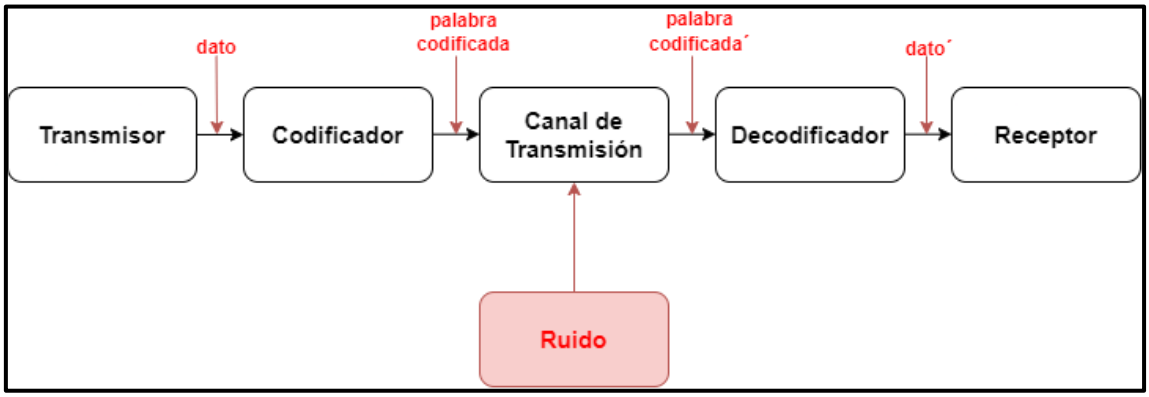

Figura 25. Proceso de codificación y decodificación de la información.

Los Códigos Detectores de Errores tienen la capacidad de detectar cuando un dato no es correcto (no representa una palabra codificada válida), mientras que los Códigos Correctores de Errores pueden reconstruir palabras codificadas válidas a partir de palabras con errores.

La distancia de Hamming proporciona una medida de detección y corrección de un código, la cual hace referencia al número de bits en el que difieren dos palabras. De esta forma, en un código con distancia de Hamming igual a dos, cualquier alteración de un bit

produce una palabra inválida, por lo que el error se puede detectar. Es decir, los códigos con distancia de Hamming igual a dos permiten detectar errores de un bit en los datos. Para corregir el error, hay que transformar la palabra inválida en la palabra válida más próxima. Esto sólo es posible si no hay dos o más palabras a la misma distancia de Hamming.

Para que un código detecte errores de b bits se debe cumplir que $d \geq b+1$, siendo d su distancia Hamming. Por otra parte, para que un código corrija errores en b bits, se debe cumplir que $d \geq 2b+1$ (también siendo d su distancia Hamming). Y si se requiere que un código tenga propiedades de corrección y detección, entonces $d \geq 2b_c + b_d + 1$, siendo d su distancia de Hamming, b_c los bits erróneos a corregir y b_d el número de bits erróneos adicionales a detectar.

Existen diferentes tipos de códigos Hamming, cada uno de ellos con una capacidad de detección y corrección de errores distinta. En cualquier caso, todos los códigos tienen una nomenclatura común para diferenciar entre el número total de bits de datos, número total de bits de paridad y número total de bits codificados que forman la palabra. La forma de calcular el número total de bits de paridad necesarios para Hamming se define de la siguiente forma:

- d = mínimo número de bits de paridad.
- b = número total de bits de datos.
- t = número total de bits que conforman la palabra codificada.
- Calculo bits paridad mínimos → $2^d \geq d + b + 1$
- Calculo bits totales codificados → $t = d + b$
- Calculo matriz de paridad → $t * d$

Una vez que se ha obtenido el número mínimo de bits de paridad necesario, se pueden definir la cantidad total de bits *t*. Por ejemplo, en el caso de necesitar proteger cuatro bits de datos, serán necesarios un total de 3 bits de paridad, ya que la distancia de Hamming calculada es igual a 3, lo cual conformaría el conocido código Hamming (7,4), siendo la matriz de paridad de un tamaño 7x3. Obtenidos dichos datos, se puede mostrar la matriz de paridad resultante y las posiciones que ocupará cada bit, siendo las posiciones de potencia de dos ocupadas por los bits de paridad (*p*), y el resto para las de datos (*d*), tal y como se muestra en la figura 26.

Para obtener las fórmulas para la codificación de los datos, en primer lugar, se numeran los bits de la palabra codificada del 1 al 7 y se representan en binario. Los bits cuya posición representada en binario tengan un solo ‘1’ corresponden a bits de código, y los demás son los bits de datos. Cada bit de código se obtiene calculando la paridad del resto de bits cuya posición representada en binario tiene a ‘1’ el mismo bit que el bit de código. Por ejemplo, el bit de código en la posición 001 se obtiene calculando la paridad de los bits de datos situados en las posiciones 011, 101 y 111, o el bit de código en la posición 010 se obtiene con los bits 011, 110 y 111.

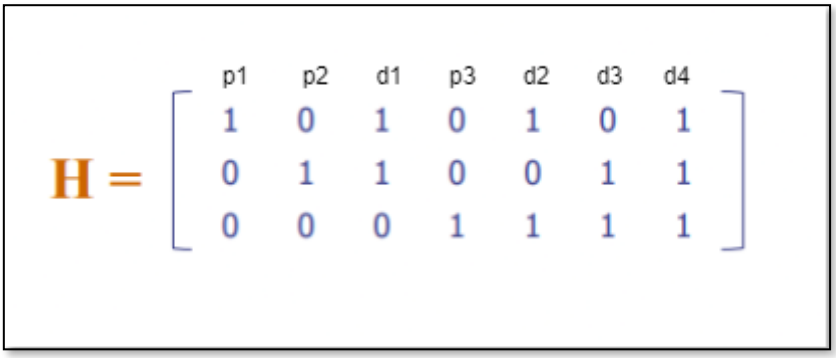

Figura 26. Matriz de paridad SEC Hamming (7,4).

Veamos ahora un ejemplo de codificación y decodificación de datos mediante este ECC. Para ello, es necesario expresar las posiciones de los bits en binario para comprobar que bits de datos comprueba cada bit de paridad, aunque puede verse reflejado en la matriz de paridad mostrada anteriormente.

En este caso la posición del bit de paridad determinará la secuencia de bits que alternativamente comprueba y salta. Mediante una operación XOR entre los bits de datos que comprueba se obtendrá el valor del bit de paridad asociado que junto a los bits de datos conformarán la palabra codificada.

- *p1* = *d1* XOR *d2* XOR *d4*
- *p2* = *d1* XOR *d3* XOR *d4*
- *p3* = *d2* XOR *d3* XOR *d4*

	p1	p2	d1	p3	d2	d3	d4
Palabra de datos(sin paridad)			0		1	1	0
Paridad 1	1		0		1		0
Paridad 2		1	0			1	0
Paridad 3				0	1	1	0
Palabra de datos codificada	***1***	***1***	***0***	***0***	***1***	***1***	***0***

Figura 27. Diseño codificador SEC Hamming (7,4).

De esta forma se obtiene la palabra codificada (1100110). Se añade a esta palabra codificada un error en un bit de forma que la palabra recibida por el receptor es 1100100, donde se resalta el bit erróneo. Si ahora se decodifica obteniendo de nuevo los valores para los tres bits de paridad, se obtiene el síndrome (110). Al ser este diferente de cero se asume que el dato no es correcto. En la matriz de paridad cuyo valor coincida con el síndrome obtenido se indica la posición de bit erróneo y su corrección se basa en invertir el bit correspondiente.

	p1	p2	d1	p3	d2	d3	d4
Palabra de datos codificada	1	1	0	0	1	1	0
Palabra de datos con error de un bit inyectado	1	1	0	0	1	0	0
Paridad 1	1		0		1		0
Paridad 2		0	0			0	0
Paridad 3				1	1	0	0

Figura 28. Diseño decodificador SEC Hamming (7,4).

	Paridad calculada	Paridad recibida	Síndrome
Paridad 1	1	1	0
Paridad 2	0	1	1
Paridad 3	1	0	1

Figura 29. Cálculo del síndrome SEC Hamming (7,4).

Con este método se detecta en que bit se ha producido el error y se obtiene de esta palabra codificada corregida la cadena de bits de datos enviados (*d1*, *d2*, *d3*, *d4*) por el nodo emisor mediante el canal de comunicación.

Una vez sentadas las bases sobre el código Hamming (7,4) se describe a continuación la ampliación de este código utilizado como uno de los elegidos durante la realización del proyecto y que ampliará las capacidades de detección de errores ya que implementa la capacidad de detección de errores dobles adyacentes.

El código Hamming extendido (8,4) es una ampliación del código Hamming (7,4) que añade a la palabra codificada un nuevo bit de paridad. Este nuevo bit de paridad será el resultado de una operación XOR entre la cadena de siete bits resultante ya mencionada y será incluido en la posición MSB (*Most Significative Bit*) pasando ahora a tener 8 bits.

Una vez codificado el mensaje de datos mediante el código Hamming (7,4) y añadido este nuevo bit de paridad total a la cadena, esta se envía por el canal de comunicación y procede exactamente igual que en la decodificación del código Hamming (7,4). Pero en este caso, además del síndrome resultante que permita descubrir el bit erróneo, también se compara el bit de paridad total enviado por el nodo emisor y el bit de paridad total calculado en el nodo receptor una vez recibida la palabra codificada, obtenido de la misma forma. Por tanto, ahora se tienen dos valores para detectar errores dobles adyacentes, errores simples y corregir errores simples: el síndrome y el bit de paridad total.

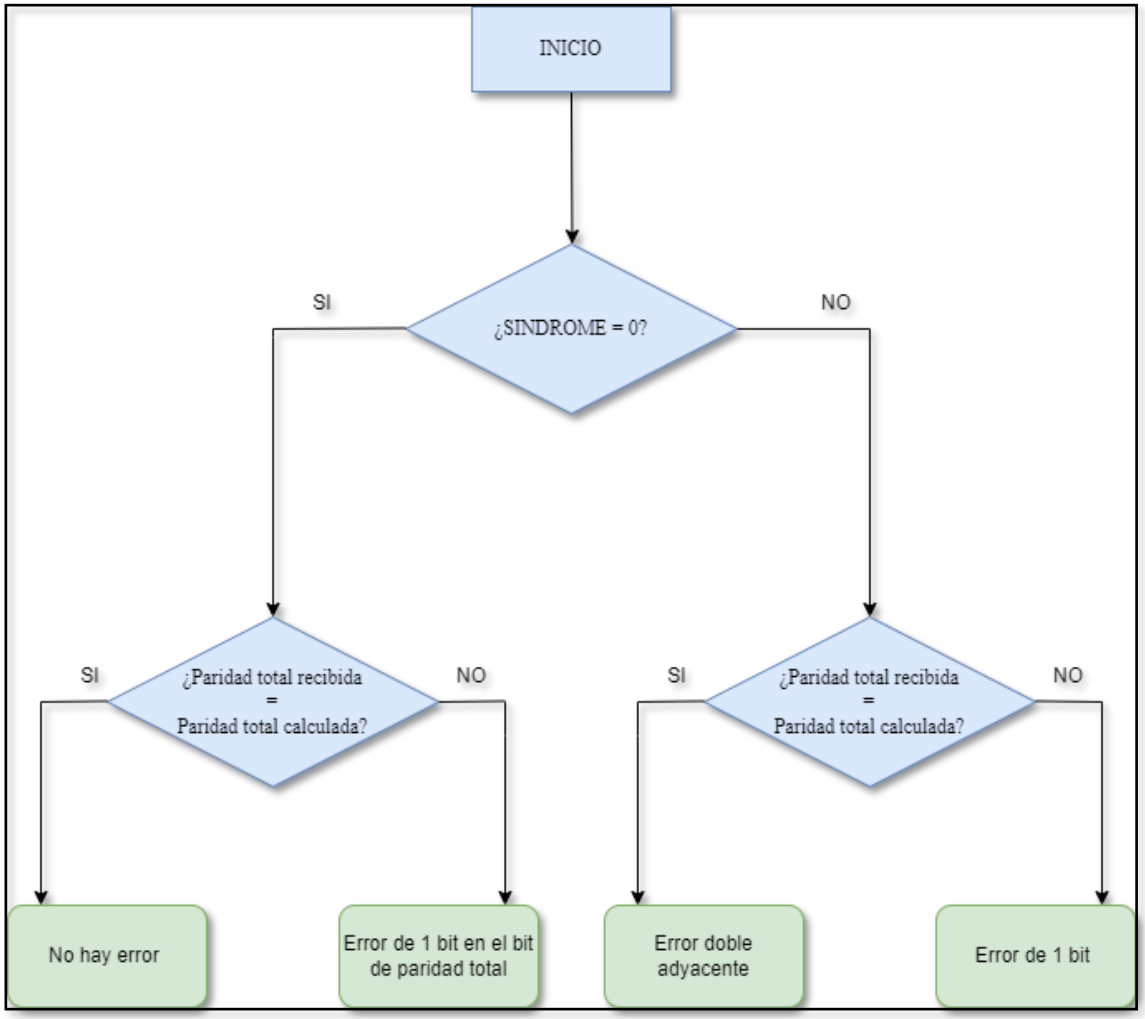

Figura 30. Diagrama de flujo de la decodificación con código SEC Hamming (7,4).

Una vez visto el diagrama se observa como en función de los valores del síndrome y de la comparación mediante una operación XOR del bit de paridad total recibido y del bit de paridad total calculado, la detección y/o corrección es distinta. Cuando el síndrome y la operación entre las paridades totales tiene un valor igual a 0 no existe ningún error en la palabra codificada recibida por lo que se puede continuar con las operaciones consiguientes. De la misma forma se puede continuar si además del valor 0 del síndrome el valor de la paridad total comparada es diferente de 0. Cuando el valor del síndrome es diferente de cero se debe prestar atención al valor de la paridad total para considerar un error de un solo bit (corregible) y un error doble adyacente, el cual obligará al nodo receptor a pedir el reenvío de la palabra codificada al nodo emisor.

En definitiva, el código Hamming extendido permite añadir una nueva funcionalidad y una capacidad de detección de errores dobles adyacentes aumentando así su capacidad de tolerancia a fallos mediante la adición de un nuevo bit de paridad total.

4.2 Códigos de corrección de errores utilizados

Se describen a continuación detalladamente las definiciones de cada uno de los ECC escogidos en el presente TFM para comprobar la tolerancia a fallos entre los vehículos utilizados. Se ha elegido una longitud total de 8 bits pues esta es la longitud de los datos a enviar por Bluetooth. De esta forma, se pueden codificar los diferentes comandos con cuatro bits, y utilizar otros cuatro bits para protección. Con este formato, es posible aumentar el repertorio de comandos si fuese necesario. Así pues, los ECC utilizados en este trabajo y que nos permitirán comprobar la viabilidad de la propuesta son los siguientes:

- Código SEC-DED Hamming (8,4)

El código de Hamming es un código SEC (*Single Error Correction*), que es capaz de corregir un bit erróneo con operaciones de codificación y decodificación sencillas y rápidas, así como con la redundancia más baja. La distancia de Hamming de estos códigos es 3. Este ECC es utilizado ampliamente en sistemas de comunicación y almacenamiento de datos para garantizar la integridad de la información transmitida y/o almacenada. La matriz de paridad para la versión (7, 4) de este ECC (4 bits de datos codificados en una palabra de 7 bits) se muestra a continuación:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Se puede aumentar la tolerancia a fallos de este ECC añadiendo un bit de paridad extra. De esta forma, el código de Hamming extendido permite corregir un error en un bit, o detectar dos bits en error. El bit de paridad extra añadido se calcula como la paridad par de toda la palabra codificada. Al extender el código con este bit, se aumenta la distancia de Hamming a 4, lo que permite la cobertura indicada. De esta forma, se dice que los códigos de Hamming extendidos son códigos SEC-DED (*Single Error Correction – Double Error Detection*).

- Código SEC-DED Hsiao (8,4)

Los códigos Hsiao son una versión optimizada de los códigos de Hamming extendidos. En este caso, la matriz de paridad de un código Hsiao debe cumplir los siguientes requerimientos:

- Cada columna tiene que ser diferente y distinta de cero.
- Cada columna contiene un número impar de 1s.
- El número total de 1s en la matriz de paridad debe ser mínimo.
- El número de 1s de cada fila debe ser idéntico, o lo más cercano posible, al número medio de 1s.

La matriz de paridad utilizada en este trabajo es la siguiente:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

El resultado de diseñar un código con estas características es que los errores simples generarán síndromes únicos, y con un número impar de unos. Por el contrario, los errores dobles generan síndromes con un número par de unos, que además se pueden repetir (distintos errores dobles pueden generar el mismo síndrome). De esta forma, se puede distinguir fácilmente entre errores simples y dobles. Los primeros se pueden corregir y los segundos solo se pueden detectar.

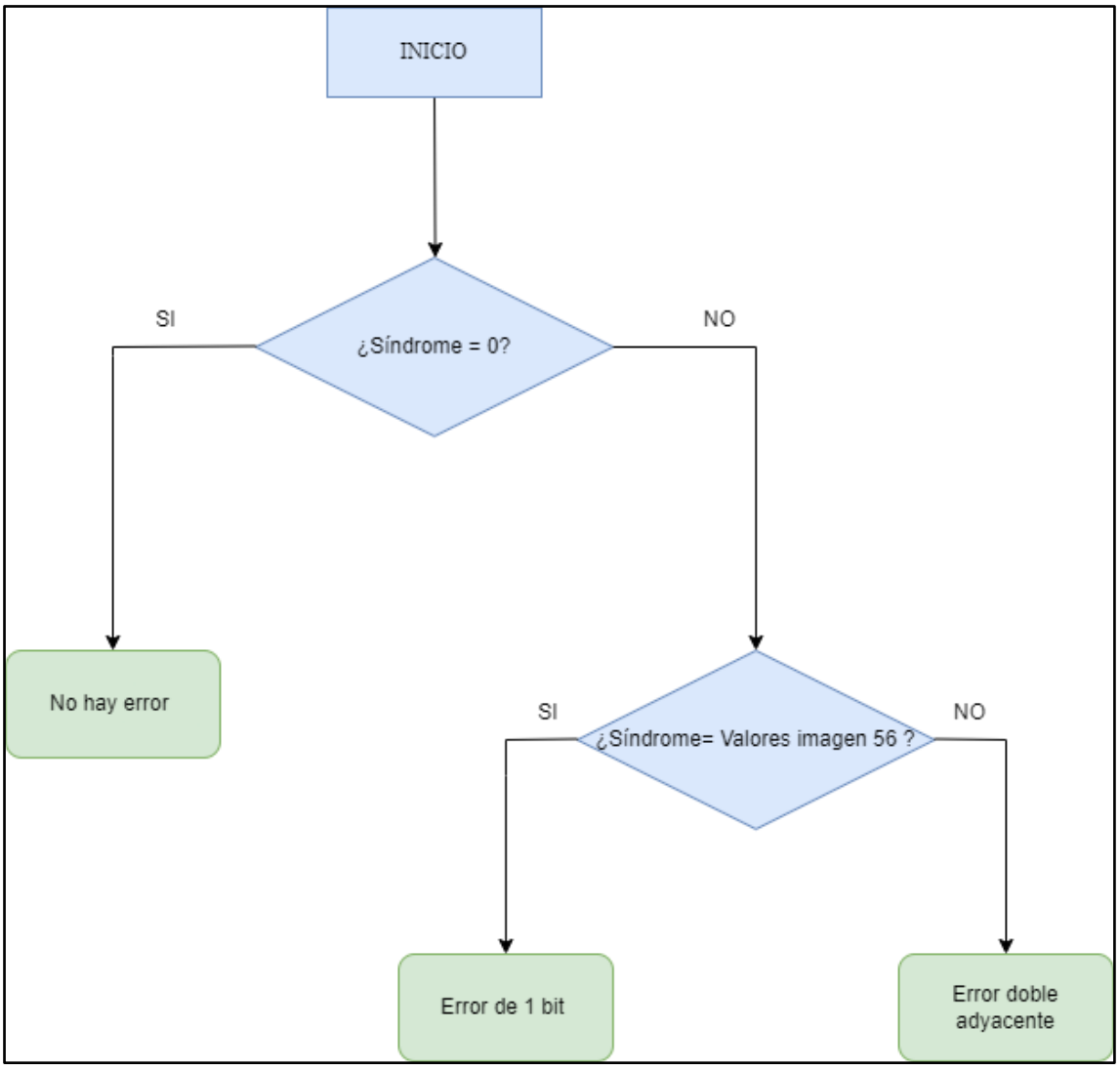

Figura 31. Diagrama de flujo de la decodificación con código SEC-DED Hsiao (8,4).

Siguiendo el diagrama anterior si el síndrome calculado en el dispositivo receptor es igual a cero la palabra codificada no tiene ningún error. Si el valor es distinto de cero se debe consultar que valor es para diferencia entre errores simples que se puede corregir o

errores dobles adyacente que suponen una petición de reenvío de la palabra codificada al dispositivo emisor.

En definitiva, el código Hsiao permite la detección de errores simples y dobles adyacentes además de poder corregir los simples. En ciertos aspectos es muy parecido al código Hamming extendido lo cual permite su comparación de prestaciones en los siguientes capítulos.

- Código SEC-DAEC-TAED (8,3)

El ultimo código implementado en este trabajo ha sido creado específicamente por el autor del proyecto mediante una herramienta del GSTF. El código recibe el nombre de SEC-DAEC-TAED, creado e implementado expresamente para este proyecto. De esta forma, se aplican los conocimientos obtenidos durante el máster cursado para implementar un código de detección y corrección de errores que cubre las necesidades del proyecto.

El nombre recibido reside en la funcionalidad implementada para este código:

- SEC (*Single-Error-Correction*): capacidad de detectar y corregir errores simples o de un solo bit.

- DAEC (*Double-Adjacent-Error-Correction*): capacidad de detectar y corregir errores dobles adyacentes.

- TAED (*Double-Adjacent-Error-Detection*): capacidad de detectar errores triples adyacentes.

	¿Detectables?	¿Corregibles?
Errores simples	Si	Si
Errores dobles adyacentes	Si	Si
Errores triples adyacentes	Si	No
Errores de mas de tres bits	No	No

Figura 32. Tabla de corrección de errores del código SEC-DAEC-TAED (8,3).

Esta implementación es similar en algunas funcionalidades a Hsiao y Hamming, los cuales son capaces también de detectar errores simples y dobles adyacentes. Se añade la capacidad de poder corregir también los errores dobles adyacentes, así como detectar ahora los triples adyacentes.

La idea principal del diseño de este ECC ha sido tener una tolerancia a fallos diferente a los dos ECC anteriores, siempre teniendo en cuenta que el tamaño máximo de la palabra codificada es de 8 bits. En concreto, queríamos reforzar la corrección y detección de errores adyacentes. Se define el error adyacente como un error múltiple donde todos los bits erróneos son contiguos, siendo este el tipo de error múltiple más frecuente. El problema que hemos encontrado es que no existe un ECC capaz de corregir dos errores adyacentes con cuatro bits de paridad para cuatro bits de datos. Así pues, hemos optado por diseñar un ECC para tres bits de datos y cinco bits de paridad. Con este nuevo requerimiento, y utilizando nuestra metodología, sí que hemos podido diseñar un ECC capaz de corregir errores simples y dobles adyacentes, así como de detectar errores triples adyacentes (SEC: *Single Error Correction* – DAEC: *Double Adjacent Error Correction* – TAED: *Triple Adjacent Error Detection*).

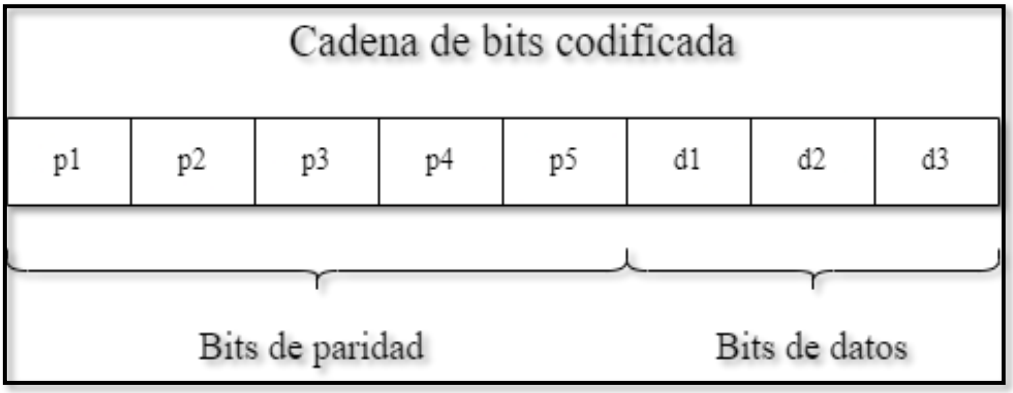

Figura 33. Posiciones de los bits de la palabra codificada.

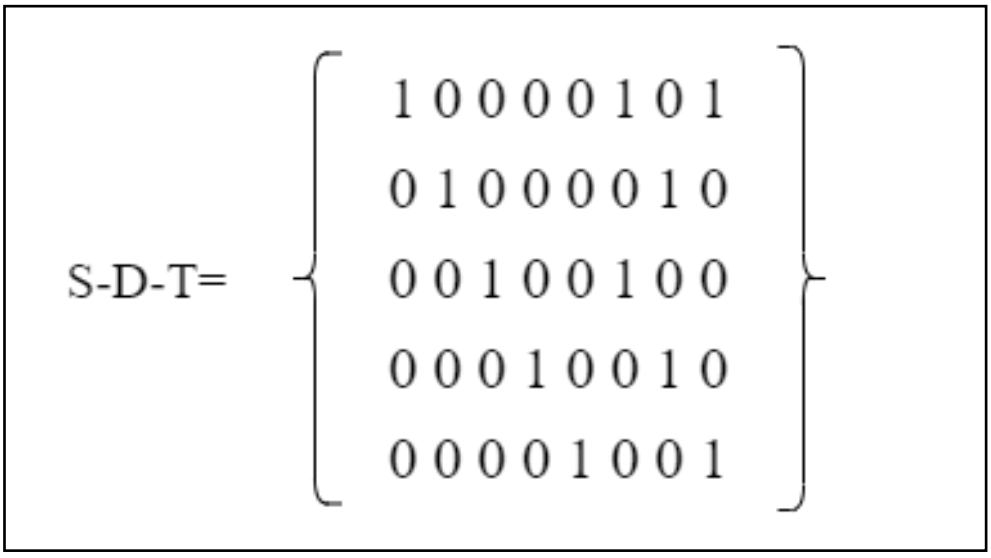

Figura 34. Matriz de paridad del código SEC-DAEC-TAED (8,3).

En las imágenes anteriores se muestran, por un lado, la cadena de bits codificada que será transmitida y cómo están distribuidos sus bits, y por otro lado, la matriz

resultante de 5x8. Los tres bits de datos se colocan en las posiciones menos significativas mientras que los bits de paridad se colocan en las posiciones más significativas, formando la cadena de ocho bits.

Se define a continuación el cálculo de la palabra codificada de ocho bits que se enviará por el canal de comunicación. Para ello basta con expresar cómo se calculan los cinco bits de paridad y qué bits de datos comprueba.

- *p1 = d1* XOR *d3*
- *p2 = d2*
- *p3 = d1*
- *p4 = d2*
- *p5 = d3*

Con los cinco bits de paridad calculados y los datos que se requieren enviar, se conforma la nueva palabra de ocho bits codificada y que posteriormente deberá ser decodificada por el dispositivo receptor para obtener los datos. En esta decodificación, se podrán detectar errores simples, dobles adyacentes y triples adyacentes mientras que únicamente podrán corregirse errores simples y dobles adyacentes.

A continuación, y de forma gráfica, se muestra un ejemplo en el que los datos a enviar requeridos deben ser codificados primero. Para ello, los cinco bits de paridad comprueban una serie de bits de datos específicos con lo que se obtienen los bits de código.

	p1	p2	p3	p4	p5	d1	d2	d3
Palabra de datos(sin paridad)						0	1	1
Paridad 1	1					0		1
Paridad 2		1					1	
Paridad 3			0			0		
Paridad 4				1			1	
Paridad 5					1			1
Palabra de datos codificada	***1***	***1***	***0***	***1***	***1***	***0***	***1***	***1***

Figura 35. Diseño codificador SEC-DAEC-TAED (8,3).

En el lado del dispositivo receptor, al igual que ocurre en los códigos de corrección de errores de Hamming y Hsiao, se requiere de la decodificación para detectar, y en su caso corregir, posibles errores en la palabra recibida según el tipo de errores. Mediante la comparación de los cinco bits de paridad recibida y los que se recalculan, se obtiene el síndrome con el que se observa en primera instancia si ha ocurrido una anomalía y posteriormente en qué bit o bits ha ocurrido.

	p1	p2	p3	p4	p5	d1	d2	d3
Palabra de datos codificada	1	1	0	1	1	0	1	1
Error doble adyacente inyectado	1	1	1	1	0	1	0	1
Paridad 1	0					1		1
Paridad 2		0					0	
Paridad 3			1			1		
Paridad 4				0			0	
Paridad 5					1			1

Figura 36. Diseño codificador SEC-DAEC-TAED (8,3).

Ordenando los bits de paridad mayor a menor, y convirtiéndolo a decimal, se obtiene el valor del síndrome. Dependiendo de qué valor sea el resultante, se detectan errores de un tipo, se corrigen errores de otro tipo o se requiere de una petición de reenvío al dispositivo emisor en caso de detección de error triple adyacente.

	Paridad calculada	Paridad recibida	Síndrome
Paridad 1	0	1	1
Paridad 2	0	1	1
Paridad 3	1	0	1
Paridad 4	0	1	1
Paridad 5	1	1	0

Figura 37. Cálculo del síndrome SEC-DAEC-TAED (8,3).

Si el valor obtenido en el síndrome es cero, entonces no ha ocurrido ninguna anomalía en la palabra codificada enviado, y por tanto, el dispositivo receptor puede continuar con sus funciones. En el caso de errores triples adyacentes, se detectan y se notifica al dispositivo emisor.

Para cubrir todos los errores posibles, es necesario contemplar ocho posibles casos de errores simples y siete posibles casos de errores dobles adyacentes. Los valores obtenidos por el síndrome determinarán en qué posición se ha producido cada error.

- Síndrome: 1 → Error simple en el bit 1.
- Síndrome: 2 → Error simple en el bit 2.
- Síndrome: 4 → Error simple en el bit 3.
- Síndrome: 8 → Error simple en el bit 4.
- Síndrome: 16→ Error simple en el bit 5.
- Síndrome: 5 → Error simple en el bit 6.
- Síndrome: 10 → Error simple en el bit 7.
- Síndrome: 17 → Error simple en el bit 8.
- Síndrome: 3 → Error doble adyacente en los bits 1 y 2.
- Síndrome: 6 → Error doble adyacente en los bits 2 y 3.
- Síndrome: 12 → Error doble adyacente en los bits 3 y 4.
- Síndrome: 24 → Error doble adyacente en los bits 4 y 5.
- Síndrome: 21→ Error doble adyacente en los bits 5 y 6.
- Síndrome: 27 → Error doble adyacente en los bits 6 y 7.

```
switch (sindrome_total) {
  case  0: error_estimado = 0b00000000; break;
  case  1: error_estimado = 0b10000000; break;
  case  2: error_estimado = 0b01000000; break;
  case  4: error_estimado = 0b00100000; break;
  case  8: error_estimado = 0b00010000; break;
  case 16: error_estimado = 0b00001000; break;
  case  5: error_estimado = 0b00000100; break;
  case 10: error_estimado = 0b00000010; break;
  case 17: error_estimado = 0b00000001; break;
  case  3: error_estimado = 0b11000000; break;
  case  6: error_estimado = 0b01100000; break;
  case 12: error_estimado = 0b00110000; break;
  case 24: error_estimado = 0b00011000; break;
  case 21: error_estimado = 0b00001100; break;
  case 15: error_estimado = 0b00000110; break;
  case 27: error_estimado = 0b00000011; break;
default: BTserial.write(comando); return;
}
```

Figura 38. Valores del síndrome en SEC-DAEC-TAED (8,3).

Este tercer código ha sido elaborado e implementado por el autor del documento mediante la documentación y aplicaciones de la asignatura de Adaptabilidad y Reconfiguración en Soluciones Tolerantes a Fallos y junto a los dos códigos descritos previamente, se podrán evaluar de maneras diversas la tolerancia a fallos de los vehículos. Este último ECC, además, tiene diferencias respecto al tratamiento de los diferentes tipos de errores que puedan llegar a ocurrir, tal y como se acaba de comentar.

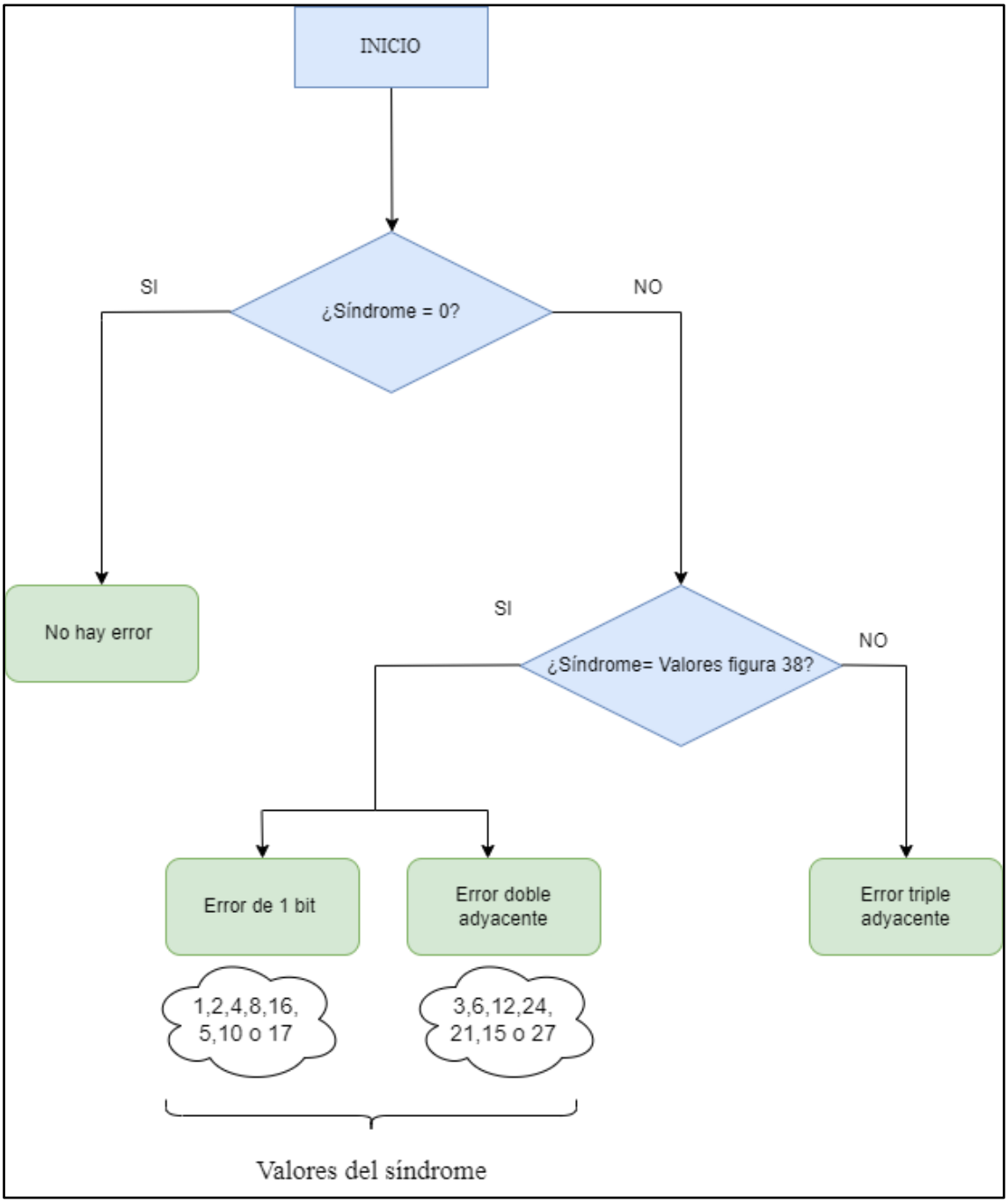

Figura 39. Diagrama de flujo de la decodificación SEC-DAEC-TAED (8,3).

- Porcentaje de cobertura SEC-DAEC-TAED (8,3)

Como estudio previo a la aplicación de este código, y dado que se ha implementado específicamente para este proyecto con nuestras propias metodologías, hay que analizar su cobertura para confirmar que el código se comporta de la forma esperada.

Mediante los estudios realizados para esta implementación ha sido posible obtener la métrica del porcentaje de cobertura de este ECC respecto al número de errores, tanto

aleatorios como adyacentes (bits contiguos). El número de errores contempla un número de entre cero y ocho bits.

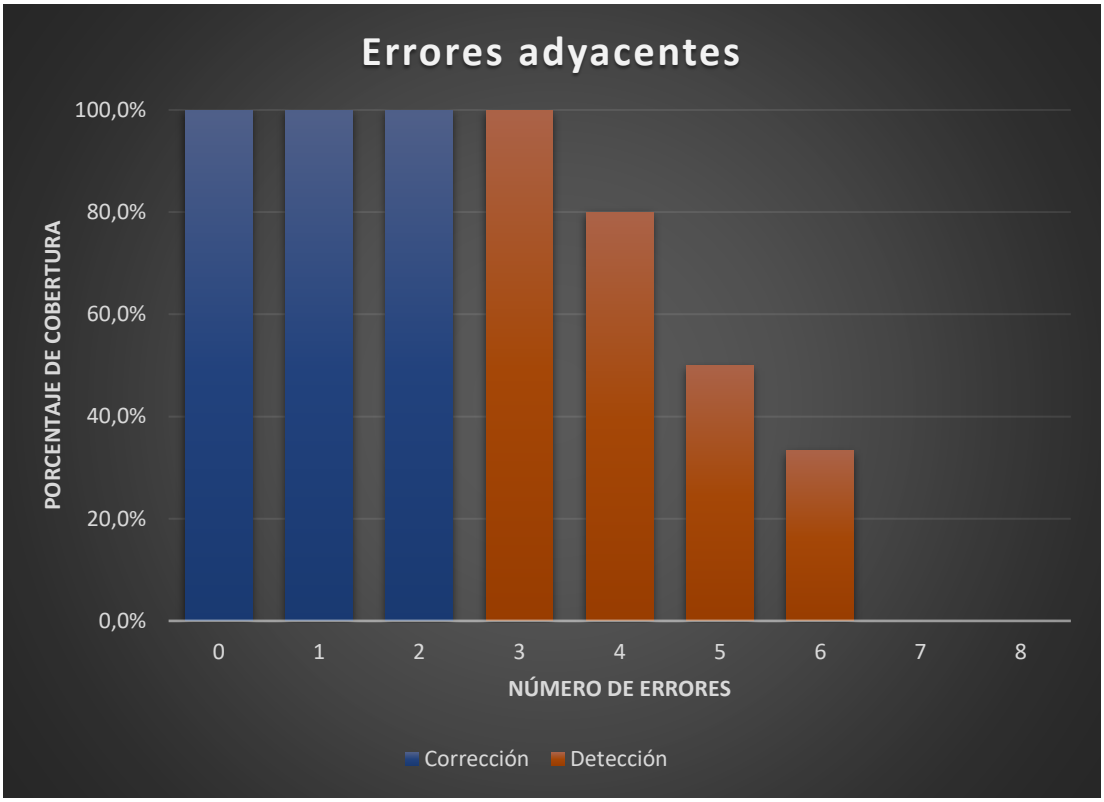

Figura 40. Porcentaje de cobertura de los errores adyacentes.

Figura 41.Porcentaje de cobertura de los errores aleatorios.

Es importante tener en cuenta que en estas gráficas no se considera la frecuencia con la que aparecen los distintos tipos de error. Lo que se hace es comprobar todos los posibles errores de un tipo, y determinar qué porcentaje de los mismos es corregido o detectado. Como era de esperar, el código funciona correctamente cuando no se produce ningún error. Esto es importante, hay que asegurarse de que sea así. Aunque, como se ha dicho, no se entra a valorar la frecuencia, este será el caso más frecuente. También es capaz de detectar y corregir todos los errores de un solo bit. Estos dos casos son comunes en ambas gráficas.

Las diferencias vienen en los errores múltiples, donde se distingue entre aleatorios (todas las posibles combinaciones) y adyacentes (solo los errores múltiples donde los bits erróneos sean contiguos). Por ejemplo, hay 28 combinaciones distintas de errores de dos bits en un dato de ocho bits. Como puede observarse en la gráfica de errores aleatorios, se corrige el 25%, es decir, solo 7 de esos 28 casos. Efectivamente, solo hay 7 combinaciones de errores dobles adyacentes en un dato de 8 bits. Dado que el código está preparado para corregirlos, son esas las combinaciones que suponen el 25% en la gráfica de errores aleatorios y, a la vez, suponen el 100% de errores adyacentes de dos bits.

Los errores triples son detectados al 100% si son adyacentes mientras que si son aleatorios esta capacidad de detección es de en torno al 61%, sin tener la posibilidad de corregirlos. A partir de aquí, se producen porcentajes de detección descendentes conforme aumenta el número de errores. Obviamente, el código no está diseñado para encontrarse con esas situaciones, por lo que es un comportamiento esperado.

En definitiva, el porcentaje de cobertura frente a errores sigue los valores asociados a la capacidad de detección y corrección del código SEC-DAEC-TAED (8,3).

5. Programación de los Códigos de Corrección de Errores

Se procede ahora a explicar de forma más detallada la implementación en los vehículos autónomos de los diferentes códigos de corrección de errores mediante el uso del entorno de desarrollo Arduino IDE. Para cada implementación del código, se desarrollan dos versiones: una en la que sólo se programa la parte de decodificación, y que servirá como protección frente a errores en la comunicación, y otra versión en la que se implementa tanto la codificación como la decodificación, y que permitirá proteger los datos durante su almacenamiento y durante su transmisión.

Como se ha mencionado a lo largo de la memoria del proyecto, se consideran únicamente cuatro datos que representan cuatro instrucciones que deben ser tratadas por el vehículo seguidor. Estas instrucciones pueden sufrir errores, tanto durante su almacenamiento como durante su transmisión, por lo que se protegen mediante los ECC descritos anteriormente. Para comprobar esta protección, se utilizará la técnica de inyección de fallos. De esta forma, se podrá estudiar el comportamiento de los vehículos en presencia de fallos, y su tolerancia a estos. Las cuatro instrucciones utilizadas son las siguientes:

- Instrucción 1 (0001): reducir velocidad.
- Instrucción 2 (0010): mantener velocidad.
- Instrucción 3 (0011): parar.
- Instrucción 4 (0100): retroceder el vehículo a velocidad reducida.

Instrucción(DEC)	Instrucción(BIN)	Definición
1	0b0001	Avanzar a velocidad inicial
2	0b0010	Reducir velocidad
3	0b0011	Detenerse
4	0b0100	Retroceder a velocidad reducida

Figura 42. Comandos implementados para los ECC.

5.1 Código de Hamming extendido SEC-DED (8,4)

- Desarrollo con codificadores

El primer código ECC a implementar es el Hamming extendido con cuatro bits de datos y cuatro de paridad, que formarán un dato codificado de 8 bits a enviar mediante el canal de comunicación Bluetooth. De esta forma, los datos a enviar se declaran como variables fijas. Estas variables incluyen tanto el dato como los bits de paridad.

En el método *setup* se inicializa el puerto serie a 38400 baudios y se establece la comunicación entre placa y módulo Bluetooth también a 38400 baudios. Del mismo modo se asigna el valor del primer dato a la variable *myByte1*. Esta asignación es igual en los tres códigos a implementar.

```
void setup()
{
  Application_FunctionSet.ApplicationFunctionSet_Init();
  BTserial.begin(38400);
  Serial.begin(38400);
  myByte1 = 0b0001;

}
```

Figura 43. Método *setup* en el código Hamming extendido (8,4) del vehículo director.

El método *codificación* requiere de la variable *myByte1*, la cual contiene el dato con el que se está trabajando en cada momento. Este método calcula los cuatro bits de cada dato y con ello los tres bits de paridad (según el código de Hamming SEC (7,4)). Mediante una operación XOR se obtiene también el último bit de paridad total necesario para la formación de la palabra codificada total de ocho bits recogidos en la variable *haming_codificado* (código de Hamming extendido SEC-DED (8,4)).

```
byte codificacion ( byte myByte1) {

  dato1 = (myByte1 >> 3) & 1; // sacamos el bit de datos 1 que nos llega
  dato2 = (myByte1 >> 2) & 1; // sacamos el bit de datos 2 que nos llega
  dato3 = (myByte1 >> 1) & 1; // sacamos el bit de datos 3 que nos llega
  dato4 = myByte1       & 1; // sacamos el bit de datos 4 que nos llega

  byte paridad1 = dato1 ^ dato2 ^ dato4;
  byte paridad2 = dato1 ^ dato3 ^ dato4;
  byte paridad3 = dato3 ^ dato2 ^ dato4;

  byte paridad_total = dato1 ^ dato2 ^ dato3 ^ dato4 ^ paridad1 ^ paridad2 ^ paridad3;

  hamming_codificado = (paridad_total << 7) | (paridad1 << 6) | (paridad2 << 5)
                     | (dato1 << 4) | (paridad3 << 3) | (dato2 << 2) | (dato3 << 1) | dato4;

  return hamming_codificado;
}
```

Figura 44. Método *codificación* en el código Hamming extendido (8,4) del vehículo director.

El método *loop* es el encargado de las tareas que se deben repetir constantemente mientras la placa Arduino esté encendida. Lo primero es hacer una llamada al método encargado de codificar los datos para que trabaje con el dato dado, y envíe esta palabra codificada por el canal de comunicación Bluetooth creado. Se implementan además dos condiciones: si el vehículo seguidor detecta un error doble adyacente pedirá un reenvío de datos mediante un comando. Ese comando se envía por el canal de

comunicación y es en esta condición donde leemos el comando (el cual se envía también codificado en 8 bits) y se decodifica para comprobar posibles errores. Una vez comprobado, ya es posible volver a tratar el dato cuya codificación tenía un error doble adyacente para volver a codificarlo y enviarlo de nuevo por el canal de comunicación. La segunda condición implementa el envío de cada uno de los cuatro datos cada tres segundos, con el fin de comprobar el correcto funcionamiento del sistema.

```
void loop()
{

  Application_FunctionSet.ApplicationFunctionSet_Tracking();
  Application_FunctionSet.ApplicationFunctionSet_SensorDataUpdate();

  byte hamming_codificado = codificacion(myByte1);
  BTserial.write(hamming_codificado);
  unsigned long tiempoActual = millis();

  if (BTserial.available() >= 1){

    byte comando;
    comando = BTserial.read(); //lectura del codigo que recibimos
    byte decodificar = decodificacion( comando );
    hamming_codificado = codificacion(myByte1);
    BTserial.write(hamming_codificado); // ewnvio de datos bien tras un error fallido

  }

  if (tiempoActual - tiempoAnterior >= 3000) { // Si han pasado 3 segundos
    tiempoAnterior = tiempoActual; // Actualizar el tiempo anterior
     myByte1 = myByte1 + 1;

  if ( myByte1 == 0b0101){

  myByte1 = 0b0001;
  }

 }

}
```

Figura 45. Método *loop* en el código Hamming extendido (8,4) del vehículo director.

Además de escribir el código del método para la codificación de los datos a enviar por el canal de comunicación, se implementa un método de decodificación para los comandos que puedan llegar por el canal desde el vehículo seguidor tras detectar un error doble adyacente. Mediante este comando, se activa la petición de reenvío al vehículo director.

Del lado del vehículo seguidor se requiere principalmente de la lectura de datos que lleguen por el canal de comunicación, un decodificador que permita detectar y corregir los errores posibles, un método para extraer los comandos de cada uno de los cuatro datos y un codificador para el comando a enviar al vehículo director en caso de detectar un error doble adyacente.

El método *loop* del código del vehículo seguidor está constantemente escuchando si se recibe alguna palabra codificada por el canal de comunicación. En tal caso, esa palabra es almacenado en la variable *datohamming* para proceder a su decodificación. El valor resultante de ese método se guarda en la variable *decodificar* y se procede con los comandos del seguimiento de línea para el circuito creado.

```
void loop() {


  if( BTserial.available() >= 1){


     datohamming = BTserial.read();
     byte decodificar = decodificacion( datohamming );
     lectura();
     switch(estado){
     case 1:{
      servo2.write(90);
      delay(30);
      break;
  }
    case 2:{
      servo2.write(45);
      delay(30);
      break;
  }
    case 3:{
      servo2.write(135);
      delay(30);
      break;
    }
  }
```

Figura 46. Método *loop* en el código Hamming extendido (8,4) del vehículo seguidor.

El método decodificación es el encargado de la detección de errores dobles adyacentes y de errores simples, además de corregir estos últimos.

```
byte decodificacion ( byte hamming_recibido) {

  int solucion;
  byte paridad_recibida = (hamming_recibido >> 7) & 0b0001; // sacamos el bit de paridad total que nos llega
  byte paridad1 = (hamming_recibido >> 6) & 0b0001; // sacamos el bit de paridad 1 que nos llega
  byte paridad2 = (hamming_recibido >> 5) & 0b0001; // sacamos el bit de paridad 2 que nos llega
  byte paridad3 = (hamming_recibido >> 3) & 0b0001; // sacamos el bit de paridad 3 que nos llega

  dato1 = (hamming_recibido >> 4) & 0b0001; // sacamos el bit de datos 1 que nos llega
  dato2 = (hamming_recibido >> 2) & 0b0001; // sacamos el bit de datos 2 que nos llega
  dato3 = (hamming_recibido >> 1) & 0b0001; // sacamos el bit de datos 3 que nos llega
  dato4 = hamming_recibido  & 0b0001; // sacamos el bit de datos 4 que nos llega
```

Figura 47. Técnica de decodificación ECC Hamming extendido (8,4) en Arduino IDE en el vehículo seguidor.

El proceso de decodificación y comprobación de errores es el siguiente:

- En primer lugar, a partir de los bits de datos recibidos, se recalculan todos los bits de paridad (los parciales y el bit de paridad total).
- A continuación, se calcula el síndrome, comparando para ello los bits de paridad parcial calculados con los recibidos. También se compara el bit de paridad total recibido y calculado.

```
byte paridad_real = dato1 ^ dato2 ^ dato3 ^ dato4 ^ paridad1 ^ paridad2 ^ paridad3;

byte p1_corregida = dato1 ^ dato2 ^ dato4;
byte p2_corregida = dato1 ^ dato3 ^ dato4;
byte p3_corregida = dato3 ^ dato2 ^ dato4;

sindrome1 = paridad1 ^ p1_corregida;
sindrome2 = paridad2 ^ p2_corregida;
sindrome3 = paridad3 ^ p3_corregida;

sindrome_total = ((sindrome3 << 2) | (sindrome2 << 1) | sindrome1) & 0b111;
paridad_total = paridad_real ^ paridad_recibida;
```

Figura 48. Cálculo del síndrome ECC Hamming extendido (8,4) en Arduino IDE en el vehículo seguidor.

- Por último, se comprueba si los bits de paridad (parcial y total) indican la presencia de algún error, y si es posible corregirlo. En el caso de la detección de un error doble adyacente, el vehículo esclavo pedirá el reenvío del mensaje mediante una variable que debe codificar antes de poder enviarla.

Síndrome_total	*Paridad_total*	*Evento*
Cero	Cero	No error
Cero	Distinto de cero	Error simple (en el bit de paridad total)
Distinto de cero	Cero	Error doble adyacente
Distinto de cero	Distinto de cero	Error simple

Figura 49. Error resultante según el síndrome y la paridad total.

La decodificación cumple la función de detectar y/o corregir errores de varios tipos. Si este método no ha detectado ningún error o ha corregido errores simples, entonces hace una llamada al método *sacar_instruccion*, cuya función es extraer los datos de la palabra codificada correcta. De esta forma, el vehículo seguidor puede interpretar las instrucciones enviadas mediante la palabra codificada por el vehículo director. Según el dato extraído, se ejecutará una de las cuatro instrucciones programadas.

- Desarrollo sin codificadores

Esta versión excluye los métodos de codificación tanto en el vehículo director para los datos que contienen los comandos a enviar como en el vehículo seguidor para el envío del comando en caso de detección de error doble adyacente.

La palabra codificada a enviar está almacenada previamente en una tabla, tal y como se puede ver a continuación. De esta forma, no es necesario codificar el dato cada vez que se quiere enviar.

Instrucción(BIN)	Instrucción (DEC)	Codigo Hamming(7 b)	Bit Paridad Total	Codigo Hamming(8 b)
0b0001	1	0b1101001	0	0b01101001
0b0010	2	0b0101010	1	0b10101010
0b0011	3	0b1000011	1	0b11000011
0b0100	4	0b1001100	1	0b11001100

Figura 50. Tabla de codificaciones de comandos para Hamming extendido (8,4).

Por el lado de la programación del vehículo seguidor la única diferencia es la exclusión del método de codificación por lo que el comando a enviar en caso de error doble adyacente se envía en este caso directamente sin ser codificado expresamente en un método de codificación, siendo este valor proporcionado por su respectiva tabla.

5.2 Código SEC-DED Hsiao (8,4)

- Desarrollo con codificadores

La segunda implementación corresponde al código de Hsiao con los cuatro bits de datos y ocho bits que formaran en su totalidad la palabra a enviar mediante el canal de comunicación Bluetooth.

En este caso, y para diversificar la programación de este ECC debido a sus similitudes con el código Hamming, se implementan de manera diferente varios aspectos relacionados con la declaración de variables y su uso.

Como ocurre también con la implementación de Hamming, las variables globales deben declararse fuera de los métodos para que puedan ser utilizadas en cualquier punto del código. En la variable *myByte1* se guarda el valor del dato en binario a utilizar en cada momento mientras que el valor de su codificación queda registrado en *hsiao_codificado.*

```
byte myByte1;
byte hsiao_codificado;
unsigned long tiempoAnterior = 0;
```

Figura 51. Definición de variables Hsiao.

La inicialización de los puertos serie, así como la comunicación entre placa y módulo Bluetooth se establecen a 38400 baudios. Se decide asignar el valor del primer dato a la variable myByte1.

```
void setup()
{
  Application_FunctionSet.ApplicationFunctionSet_Init();
  BTserial.begin(38400);
  Serial.begin(38400);
  myByte1 = 0b0001;

}
```

Figura 52. Método *setup* vehículo director Hsiao.

El método *loop* es similar al mismo método implementado para Hamming. Lo primero es la codificación mediante un método auxiliar del dato con el que se trabaje en cada momento, y cuyo resultado es el dato codificado de 8 bits a enviar. Este envío se produce cada tres segundos. Si el vehículo director recibe un dato, significa que el vehículo seguidor ha detectado un error doble en el mensaje anterior, por lo que debe codificarlo de nuevo y enviarlo otra vez al vehículo seguidor.

```
void loop()
{

  Application_FunctionSet.ApplicationFunctionSet_Tracking();
  Application_FunctionSet.ApplicationFunctionSet_SensorDataUpdate();

  hsiao_codificado = codificacion(myByte1);
  BTserial.write(hsiao_codificado);
  unsigned long tiempoActual = millis();

  if (BTserial.available() >= 1){

    byte comando;
    comando = BTserial.read(); //lectura del codigo que recibimos
    byte decodificar = decodificacion( comando );
    hsiao_codificado = codificacion(myByte1);
    BTserial.write(hsiao_codificado); // ewnvio de datos bien


  }
    if (tiempoActual - tiempoAnterior >= 3000) { // Si han pasado 3 segundos
    tiempoAnterior = tiempoActual; // Actualizar el tiempo anterior
     myByte1 = myByte1 + 1;

  if ( myByte1 == 0b0101){

  myByte1 = 0b0001;
  }
```

Figura 53. Método *loop* vehículo director Hsiao.

El vehículo se mantiene a la espera de recepción de palabras codificadas por el canal de comunicación Bluetooth. En ese caso dicha palabra es recogida mediante la variable *datoHsiao* y es decodificada mediante el correspondiente método de actuación y continúa con las instrucciones referidas al seguimiento de línea constante del circuito.

```
void loop() {


  if( BTserial.available() >= 1){

     datoHsiao = BTserial.read(); //lectura del codigo que recibimos
     byte decodificar = decodificacion( datoHsiao );

  lectura();
  switch(estado){
    case 1:{
      servo2.write(90);
      delay(30);
      break;
  }
  case 2:{
      servo2.write(45);
      delay(30);
      break;
  }
  case 3:{
      servo2.write(135);
      delay(30);
      break;
   }
  }
 }
```

Figura 54. Método *loop* vehículo seguidor Hsiao.

El método *decodificación* trabaja con el parámetro dado, el cual representa la palabra codificada recibida. Lo primero es extraer unitariamente los 4 bits de paridad recibidos y los 4 bits de datos. Se calculan localmente los bits de paridad representados por *px_corregida* para obtener los cuatro valores de cada síndrome. Mediante los valores anteriores se obtiene el síndrome total, que indica la detección y/o corrección de errores en el caso de un valor distinto de cero.

```
byte decodificacion ( byte hsiao_recibido) {

  int solucion;
  byte paridad0 = (hsiao_recibido >> 7) & 1; // sacamos el bit de paridad 0 que nos llega
  byte paridad1 = (hsiao_recibido >> 6) & 1; // sacamos el bit de paridad 1 que nos llega
  byte paridad2 = (hsiao_recibido >> 5) & 1; // sacamos el bit de paridad 2 que nos llega
  byte paridad3 = (hsiao_recibido >> 4) & 1; // sacamos el bit de paridad 3 que nos llega

  dato0 = (hsiao_recibido >> 3) & 1; // sacamos el bit de datos 0 que nos llega
  dato1 = (hsiao_recibido >> 2) & 1; // sacamos el bit de datos 1 que nos llega
  dato2 = (hsiao_recibido >> 1) & 0b0001; // sacamos el bit de datos 3 que nos llega
  dato3 = hsiao_recibido  & 1; // sacamos el bit de datos 2 que nos llega

  byte p0_corregida = dato0 ^ dato1 ^ dato2;
  byte p1_corregida = dato0 ^ dato1 ^ dato3;
  byte p2_corregida = dato0 ^ dato2 ^ dato3;
  byte p3_corregida = dato1 ^ dato2 ^ dato3;

  sindrome0 = paridad0 ^ p0_corregida;
  sindrome1 = paridad1 ^ p1_corregida;
  sindrome2 = paridad2 ^ p2_corregida;
  sindrome3 = paridad3 ^ p3_corregida;

  sindrome_total = ((sindrome3 << 3) | (sindrome2 << 2) | (sindrome1 << 1) | sindrome0) & 0b1111;
  byte error_estimado;
```

Figura 55. Método decodificación vehículo seguidor Hsiao.

Mediante el valor de síndrome total se deduce si se ha recibido una palabra codificada correcta (case 0) o si se ha detectado un error simple mediante la estructura de control *switch*, en cuyo caso tiene la capacidad de corregir invirtiendo el valor de ese bit erróneo. Si no existe ningún caso coincidente se ejecuta el bloque de código dentro de *default*, que corresponde a la detección de un error doble que no se puede corregir, por lo que se lanza la petición de reenvío al vehículo director mediante el envío del comando correspondiente.

```
switch (sindrome_total) {
  case  0: error_estimado = 0b00000000; break;
  case  1: error_estimado = 0b10000000; break;
  case  2: error_estimado = 0b01000000; break;
  case  4: error_estimado = 0b00100000; break;
  case  8: error_estimado = 0b00010000; break;
  case  7: error_estimado = 0b00001000; break;
  case 11: error_estimado = 0b00000100; break;
  case 13: error_estimado = 0b00000010; break;
  case 14: error_estimado = 0b00000001; break;

default:
byte comando = codificar(comando);
BTserial.write(comando);
Serial.println("error doble");return;
}
hsiao_recibido = hsiao_recibido ^ error_estimado;
solucion = sacar_instruccion( hsiao_recibido );
```

Figura 56. Detección y/o corrección de errores Hsiao.

Si la decodificación no ha detectado ningún error, o ha corregido un error simple, se procede con el método *sacar_instruccion* para extraer la funcionalidad requerida de cada dato enviado mediante la palabra codificada. Según el dato correspondiente se ejecutará una de las cuatro instrucciones programadas.

- Desarrollo sin codificadores

Esta versión excluye los métodos de codificación tanto en el vehículo director para los datos que contienen las instrucciones a enviar como en el vehículo seguidor para el envío del comando en caso de detección de un error doble adyacente. En lugar de hacer la codificación en cada caso, se inicializa una tabla que almacena la codificación de cada comando según el código de Hsiao (8, 4).

Instrucción(BIN)	Instrucción (DEC)	Codigo HSIAO(8 b)
0b0001	1	0b01110001
0b0010	2	0b10110010
0b0011	3	0b11000011
0b0100	4	0x11010100

Figura 57. Tabla de codificaciones de comandos para Hsiao.

En la tabla anterior se muestra cada dato expresado en binario junto a su valor en decimal y su codificación correspondiente mediante el código de Hsiao.

5.3 Código SEC-DAEC-TAED (8,3)

En cuanto al tercer y último código de corrección de errores diseñado por metodologías propias, la implementación software es más costosa que en los ejemplos anteriores. La palabra codificada está compuesta por 3 bits de datos y 5 bits de paridad, lo que suponen en su totalidad la palabra de 8 bits a enviar por el canal de comunicación.

Se definen, en primer lugar, las variables globales *codigo_sec_daec_taed* para asignar el valor del dato una vez codificado y la variable *myByte1* para almacenar el dato a trabajar en cada momento. Se inicializan el puerto serie y la comunicación serial Bluetooth a una velocidad de 38400 baudios y se asigna el valor del primer dato del conjunto a la variable *myByte1*.

```
byte myByte1;
byte codigo_sec_daec_taed1;


void setup()
{
  Application_FunctionSet.ApplicationFunctionSet_Init();
  BTserial.begin(38400); // Inicializamos el puerto serie BT (Para Modo AT 2)
  Serial.begin(38400); // Inicializamos el puerto serie
  myByte1 = 0b0001;

}
```

Figura 58 Método *setup* vehículo director SEC-DAEC-TAED (8,3).

En el método *loop* se sigue con la metodología de envío mencionada en los anteriores ECCs. El vehículo director codifica la variable *myByte1* que contiene en cada momento el dato a trabajar y lo envía por el canal de comunicación. Del mismo modo, también queda a la espera de recibir la variable comando la cual se basa en la petición de reenvío de un dato. Esta petición se produce cuando el vehículo seguidor detecta un error triple adyacente en la información recibida.

```
void loop()
{

  Application_FunctionSet.ApplicationFunctionSet_Tracking();
  Application_FunctionSet.ApplicationFunctionSet_SensorDataUpdate();
  unsigned long tiempoActual = millis();
  unsigned long tiempoAnterior = 0;

  byte codigo_sec_daec_taedl = codificacion(myBytel);
  BTserial.write(codigo_sec_daec_taedl);

  if (BTserial.available() >= 1) {

    byte comando;
    comando = BTserial.read();
    byte decodificar = decodificacion( comando );
    codigo_sec_daec_taedl = codificacion(myBytel);
    BTserial.write(codigo_sec_daec_taedl);

  }
    if (tiempoActual - tiempoAnterior >= 3000) { // Si han pasado 3 segundos
    tiempoAnterior = tiempoActual; // Actualizar el tiempo anterior
    myBytel = myBytel + 1;

  if ( myBytel == 0b0101){

  myBytel = 0b0001;
}
```

Figura 59. Método *loop* del vehículo director SEC-DAEC-TAED (8,3).

La codificación de cada dato en este ECC consta de extraer los tres bits de datos del byte almacenado en la variable *myByte1* y calcular los cinco bits de paridad a partir de los bits de datos *dato1*, *dato2* y *dato3*. Siguiendo la distribución de bits de la palabra codificada final esta se almacena en la variable correspondiente y ya se puede enviar por el canal de comunicación.

```
byte codificacion ( byte myByte1 ) {

  byte dato1 = (myByte1 >> 2) & 1; // sacamos el bit de datos 0 que nos llega
  byte dato2 = (myByte1 >> 1) & 1; // sacamos el bit de datos 1 que nos llega
  byte dato3 =  myByte1       & 1; // sacamos el bit de datos 2 que nos llega

  byte paridad1 = dato1 ^ dato3;
  byte paridad2 = dato2;
  byte paridad3 = dato1;
  byte paridad4 = dato2;
  byte paridad5 = dato3;

  byte codigo_sec_daec_taed1 = (paridad1 << 7) | (paridad2 << 6) | (paridad3 << 5) |
                               (paridad4 << 4) | (paridad5 << 3) | (dato1 << 2) | (dato2 << 1) | dato3;

  return codigo_sec_daec_taed1;
}
```

Figura 60.Método *codificación* del código SEC-DAEC-TAED (8,3).

Se inicializa también el puerto serie y la comunicación Bluetotth a 38400 baudios, así como el valor del comando a cero. En el método *loop* se espera a la recepción de una palabra codificada y en ese caso se asigna a la variable *datoSECDAECTAED* la cual se decodifica mediante otro método de decodificación implementado mientras el vehículo procede con sus instrucciones de detección de línea y movimiento por el circuito propuesto.

```
void loop() {


  if ( BTserial.available() >= 1) {

    datoSECDAECTAED = BTserial.read();
    byte decodificar = decodificacion( datoSECDAECTAED );

    lectura();
    switch (estado) {
      case 1: {
          servo2.write(90);
          delay(30);
          break;
        }
      case 2: {
          servo2.write(45);
          delay(30);
          break;
        }
      case 3: {
          servo2.write(135);
          delay(30);
          break;
        }
    }

  }
```

Figura 61.Método *loop* del vehículo seguidor en el código SEC-DAEC-TAED (8,3).

En la decodificación propia de este ECC se extraen, mediante las operaciones correspondientes, los 5 bits de paridad y los 3 bits de datos recibidos. A continuación, se calculan los nuevos bits de paridad y mediante el síndrome se compara cada par de bits de paridad recibido/calculado con el que se obtiene el valor del síndrome total para su posterior uso.

```
int solucion;
byte paridad1 = (datoSECDAECTAED >> 7) & 1;
byte paridad2 = (datoSECDAECTAED >> 6) & 1;
byte paridad3 = (datoSECDAECTAED >> 5) & 1;
byte paridad4 = (datoSECDAECTAED >> 4) & 1;
byte paridad5 = (datoSECDAECTAED >> 3) & 1;

dato1 = (datoSECDAECTAED >> 2) & 1;
dato2 = (datoSECDAECTAED >> 1) & 1;
dato3 =  datoSECDAECTAED       & 1;

byte p1_corregida = dato1 ^ dato3;
byte p2_corregida = dato2;
byte p3_corregida = dato1;
byte p4_corregida = dato2;
byte p5_corregida = dato3;
```

Figura 62. Cálculo de los bits de paridad en el código SEC-DAEC-TAED (8,3).

```
sindrome0 = paridad0 ^ p0_corregida;
sindrome1 = paridad1 ^ p1_corregida;
sindrome2 = paridad2 ^ p2_corregida;
sindrome3 = paridad3 ^ p3_corregida;
sindrome4 = paridad4 ^ p4_corregida;

byte sindrome_total = ((sindrome4 << 4) | (sindrome3 << 3) |
                       (sindrome2 << 2) | (sindrome1 << 1) | sindrome0) & 0b111111;
```

Figura 63. Cálculo del síndrome en el código SEC-DAEC-TAED (8,3).

Una vez obtenido el síndrome se aplica una estructura de control *switch* que permite evaluar su valor. Este ECC permite corregir errores de 1 bit y errores dobles adyacentes por lo que la cantidad de valores del síndrome es mayor. En el caso de la detección de un error triple adyacente se pide el reenvío de la palabra codificada mediante el envío de la variable comando, la cual es necesaria codificar primero.

```
switch (sindrome_total) {
    case  0: error_estimado = 0b00000000; break;
    case  1: error_estimado = 0b10000000; break;
    case  2: error_estimado = 0b01000000; break;
    case  4: error_estimado = 0b00100000; break;
    case  8: error_estimado = 0b00010000; break;
    case 16: error_estimado = 0b00001000; break;
    case  5: error_estimado = 0b00000100; break;
    case 10: error_estimado = 0b00000010; break;
    case 17: error_estimado = 0b00000001; break;
    case  3: error_estimado = 0b11000000; break;
    case  6: error_estimado = 0b01100000; break;
    case 12: error_estimado = 0b00110000; break;
    case 24: error_estimado = 0b00011000; break;
    case 21: error_estimado = 0b00001100; break;
    case 15: error_estimado = 0b00000110; break;
    case 27: error_estimado = 0b00000011; break;
 default:
  byte comando = codificar(comando);
  BTserial.write(comando);
  Serial.println("error triple adyacente");return;
```

Figura 64.Detección y/o corrección de errores en el código SEC-DAEC-TAED (8,3).

La decodificación permite la detección y/o corrección de ciertos tipos de errores. Si no se ha detectado ningún error, se ha corregido un error simple o se ha corregido un error doble adyacente se procede con el método *sacar_instruccion* para extraer la funcionalidad requerida de cada dato enviado mediante la palabra codificada. Según el dato correspondiente se ejecutará una de las cuatro instrucciones programadas.

- Desarrollo sin codificadores

En esta versión del código SEC-DAEC-TAED se excluyen los métodos de codificación, tanto en el vehículo director para los datos que contienen las instrucciones a enviar, como en el vehículo seguidor para el envío del comando de solicitud de reenvío, en caso de detección de un error triple adyacente. Como no se aplica la codificación en el código, se tienen que inicializar internamente mediante una tabla de valores asociados.

Instrucción(BIN)	Instrucción (DEC)	Codigo SEC-DAEC-TAED(8 b)
0b0001	1	0b10001001
0b0010	2	0b01010010
0b0011	3	0b11011011
0b0100	4	0b10100100

Figura 65. Tabla de codificaciones de comandos para el vehículo seguidor en SEC-DAEC-TAED (8,3).

En la primera columna están expresadas en binario las cuatro instrucciones posibles que puede realizar el vehículo seguidor. En la segunda su valor en decimal y en la tercera su codificación correspondiente, según el código SEC-DAEC-TAED (8,3) diseñado en este trabajo.

6. Pruebas, resultados y valoración

Una vez finalizadas las implementaciones de los códigos detectores de errores en los vehículos director y seguidor utilizando la herramienta Arduino IDE se pretende ahora valorar y analizar la mejora obtenida respecto a la confiabilidad mediante la técnica de inyección de fallos. Esta técnica permitirá comprobar el correcto funcionamiento de los ECC frente a los diferentes fallos a inyectar.

Del mismo modo, se realizan varios análisis para evidenciar los resultados en función del tamaño de software y del tiempo de ejecución de cada sistema implementado con cada ECC, tanto para la versión con codificadores como la versión sin codificadores.

Mediante el circuito de pruebas propuesto (en la imagen), se analiza el comportamiento del sistema en función del código corrector de errores implementado en cada momento y del tipo de errores inyectados. Este circuito compone una especia de óvalo en el que los dos vehículos autónomos deben seguir la línea negra durante todo el trayecto mientras el vehículo seguidor ejecuta además las operaciones enviadas por el vehículo director.

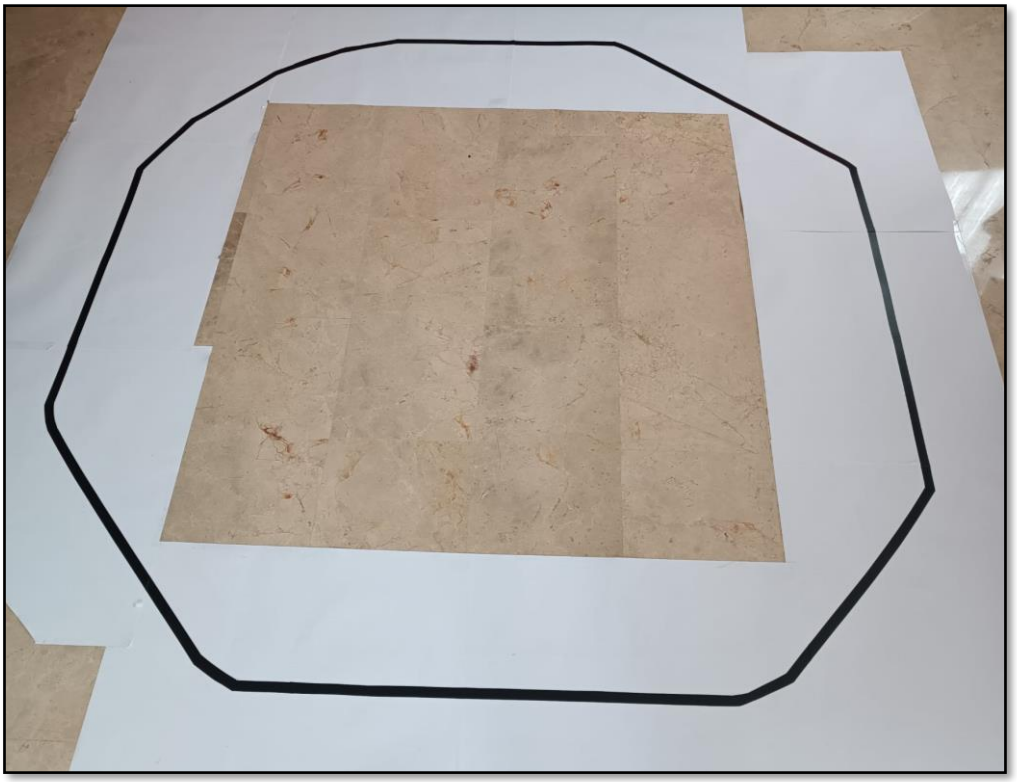

Figura 66. Circuito de pruebas propuesto.

Para comprobar la detección y/o corrección de errores se inyectan los fallos de manera sistemática. En el caso de los errores simples se inyectan en cada uno de los

ocho bits que componen la palabra codificada a enviar en cada caso. Por lo que, una vez codificado el dato a trabajar u obtenido de la tabla relacionada, si implementamos la versión sin codificadores, se inyectan fallos controlados en un bit diferente cada vez que dicha palabra codificada se envía con tal de comprobar todas las posibilidades. En el caso de la inyección de errores dobles adyacentes de manera sistemática se tienen un total de siete casos posibles, para los triples adyacentes se tienen seis posibles casos y cinco para los cuádruples adyacentes.

La transmisión de los datos codificados se realiza de manera continua en un bucle. Sin embargo, el dato a enviar cambia cada tres segundos. Las pruebas consisten en enviar, de forma cíclica, las cuatro instrucciones, en orden secuencial, comenzando por el comando uno hasta el cuatro. Una vez enviado el cuarto comando el proceso vuelve a empezar desde el primero, repitiendo el ciclo. De esta forma, se comprueban las instrucciones individualmente, con un tiempo suficiente para ver el comportamiento en el circuito propuesto.

Instrucción(DEC)	Instrucción(BIN)	Definición
1	0b0001	Avanzar a velocidad inicial
2	0b0010	Reducir velocidad
3	0b0011	Detenerse
4	0b0100	Retroceder a velocidad reducida

Figura 67. Definición de los comandos.

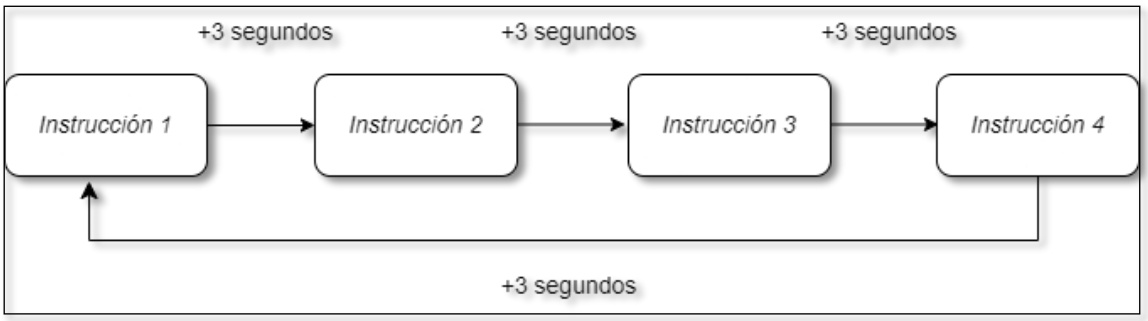

Figura 68. Iteración de envío de los cuatro comandos.

Lanzar las cuatro instrucciones de forma cíclica permite, además de probar todos los comandos, evitar que el vehículo seguidor (más rápido) alcance al vehículo director, por las configuraciones de cada vehículo y la manera en la que toman las curvas. Se propone poner en marcha los vehículos con una duración de aproximadamente 1,5 o 2 vueltas, las cuales son suficientes para analizar el sistema en cada caso.

El comportamiento correcto del sistema puede observarse en el vídeo disponible en este enlace: https://youtu.be/ymK4kNL-A9E

6.1 Inyección de fallos

En primer lugar, se deben especificar los diferentes tipos de errores con los que se va a trabajar y qué capacidad tiene cada ECC para detectar y/o corregir el error para poder continuar con sus instrucciones. La hipótesis de fallo con la que se trabaja en este proyecto es que los errores múltiples, en caso de que se produzcan, tendrán una causa física común y afectarán a bits contiguos. Por ello, el estudio se enfoca en los errores simples y múltiples adyacentes, mucho más frecuentes que los múltiples no adyacentes.

	SEC-DED Hamming (8,4)	SEC-DED Hsiao (8,4)	SEC-DAEC-TAED (8,3)
Errores simples	*Detectable* *Corregible*	*Detectable* *Corregible*	*Detectable* *Corregible*
Errores dobles adyacentes	*Detectable* *No corregible*	*Detectable* *No corregible*	*Detectable* *Corregible*
Errores triples adyacentes	*No detectable* *No corregible*	*No detectable* *No corregible*	*Detectable* *No corregible*
Errores cuádruples adyacentes	*No detectable* *No corregible*	*No detectable* *No corregible*	*No detectable* *No corregible*

Figura 69. Capacidad de detección y/o corrección de cada ECC.

- Sistema sin ECC

El primer caso de estudio permite la comprobación del comportamiento del sistema cuando no existe ningún código de corrección de errores. Los datos se envían directamente por el canal de comunicación sin que estos sean codificados. En cuanto el dato es recibido se procesa para extraer la instrucción correspondiente y por lo tanto no se decodifica.

En este sistema no se implementa ningún método adicional en los códigos fuente del vehículo director y seguidor por lo que la inyección de fallos de cualquier tipo provoca comportamientos no deseados o erróneos.

El código fuente del vehículo maestro añade una variable que representa el error a inyectar en cada caso. Primero se trabaja con los errores simples de manera sistemática para la comprobación bit a bit. Una vez comprobado se inyectan los errores dobles, triples y cuádruples adyacentes todos ellos de manera sistemática.

```
void loop()
{
  delay(10);
  Application_FunctionSet.ApplicationFunctionSet_Tracking();
  Application_FunctionSet.ApplicationFunctionSet_SensorDataUpdate();
  myByte1 = myByte1 ^ error_inyectado;
  BTserial.write(myByte1);
```

Figura 70. Inyección de fallos mediante Arduino IDE.

Como este método no incluye la posibilidad de detección y/o corrección de errores el comportamiento del vehículo seguidor es erróneo o nulo para cualquier tipo de inyección.

En el caso de la inyección de errores triples y cuádruples adyacentes el vehículo seguidor se queda estático en la posición inicial en el que se coloca, lo que conlleva que en unos segundos el vehículo director colisione con él. Sin embargo, cuando la inyección es de errores simples o errores dobles adyacentes ejecuta durante unos segundos la instrucción uno y la instrucción cuatro respectivamente, siendo estos comportamientos no deseados y en los cuales también los vehículos terminan colisionando.

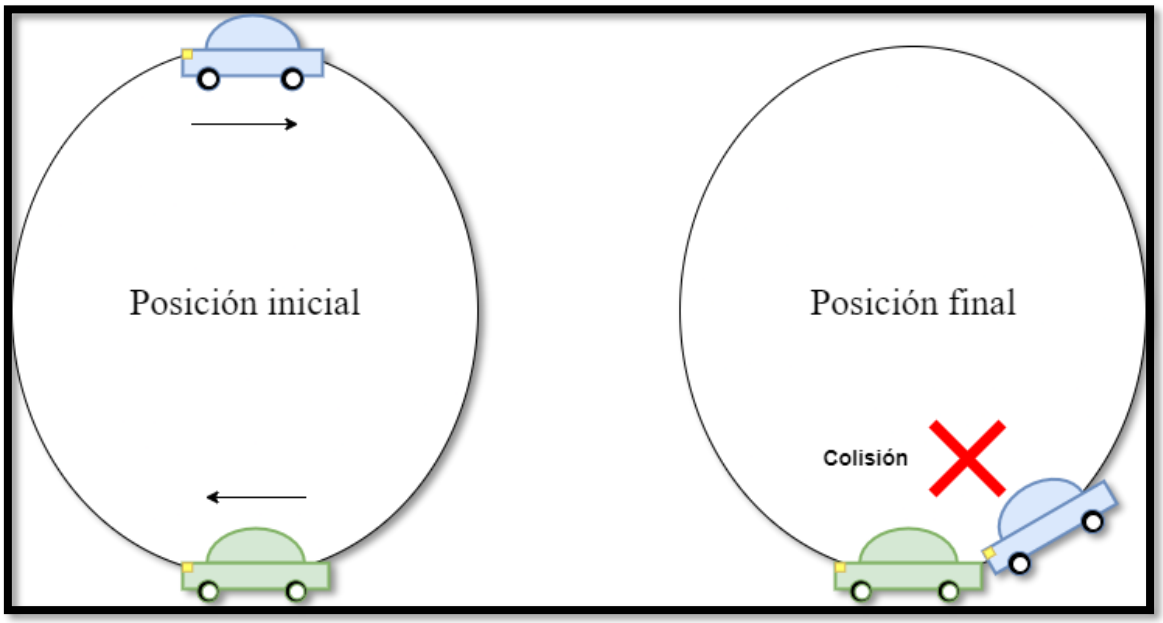

Figura 71. Comportamiento erróneo en el sistema sin ECC.

En definitiva, utilizando este método para el envío y la recepción de datos mediante un canal de comunicación asíncrono bidireccional provoca una tolerancia a

fallos nula, debido a que no hay detección ni corrección de errores, dando lugar a comportamientos no deseados como el mostrado en la imagen anterior.

- Sistemas con códigos SEC-DED (8,4)

El segundo caso de estudio incorpora al sistema los ECCs de Hamming o Hsiao, los cuales corrigen fallos simples de un bit y pueden detectar los errores dobles. Por ello, la inyección de estos dos tipos de fallos debe provocar un comportamiento normal y esperado cuando colocamos los vehículos en el circuito propuesto.

Como es de esperar, el vehículo seguidor recibe los datos codificados con errores simples y errores dobles adyacentes. A pesar de ello, es capaz de extraer las instrucciones correctas, por lo que este vehículo realiza la instrucción número uno hasta la cuatro cada tres segundos, repitiendo el bucle hasta completar aproximadamente una vuelta y media al circuito. El comportamiento es el esperado, totalmente correcto.

En cuanto al resto de errores inyectados, el comportamiento es diferente en cada caso dependiendo también del ECC con el que se trabaje. Cuando el vehículo seguidor recibe una palabra codificada con una inyección de fallos triple adyacente en su cadena de bits, este no es capaz de detectarlo y por tanto su comportamiento es erróneo. En el caso del código Hamming (8,4) el sistema ejecuta las tres primeras instrucciones, pero no ejecuta la última de ellas provocando la colisión de los vehículos. Cuando es el código Hsiao (8,4) es el implementado y se inyectan errores triples adyacentes el vehículo seguidor es capaz únicamente de ejecutar las instrucciones número tres y cuatro, causando finalmente que el vehículo director le alcance y colisione con él ejecutando únicamente y por error la instrucción número tres que indica que pare los motores y la instrucción cuatro que indica que avance hacia atrás a una velocidad reducida provocando una colisión final. Todos estos comportamientos anómalos son esperables, ya que estos códigos no están preparados para tales situaciones, haciendo que su comportamiento sea imprevisible, entregando mensajes con una instrucción diferente a la original. Es por ello que algunas veces no se ejecuta ninguna instrucción y otras ejecuta otra no pedida en ese momento. En el caso de los errores cuádruples en ambos ECCs el vehículo seguidor se mantiene estático en la posición inicial, provocando también una colisión cuando el vehículo director le alcanza.

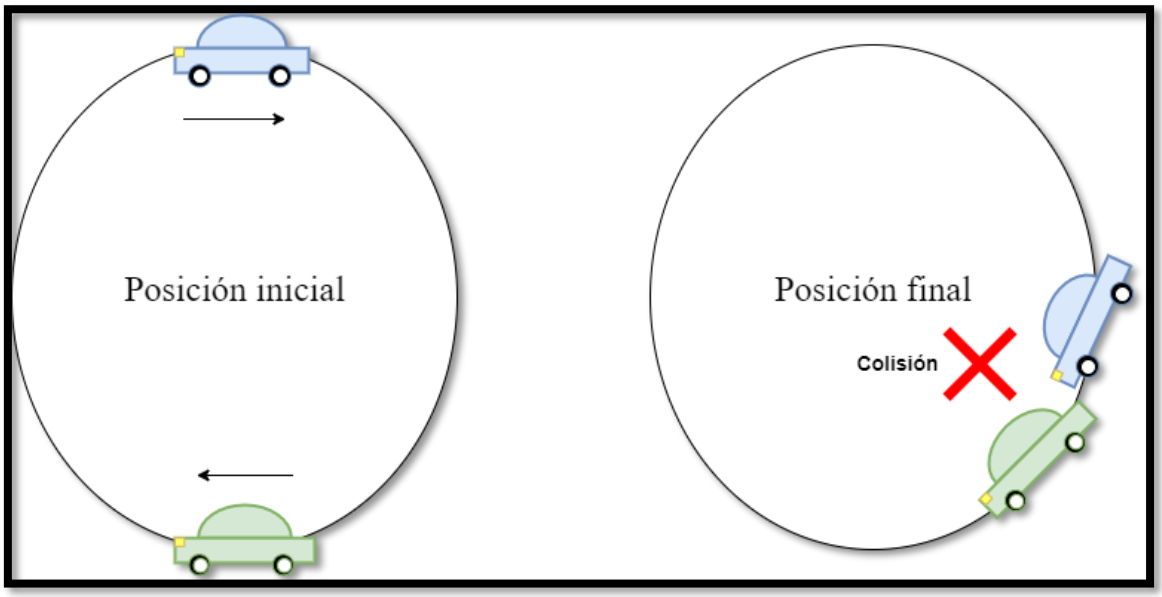

Figura 72. Comportamiento erróneo en el sistema con código SEC-DED Hsiao (8,4).

Prueba 3: https://youtu.be/MDOWqlgiMWM

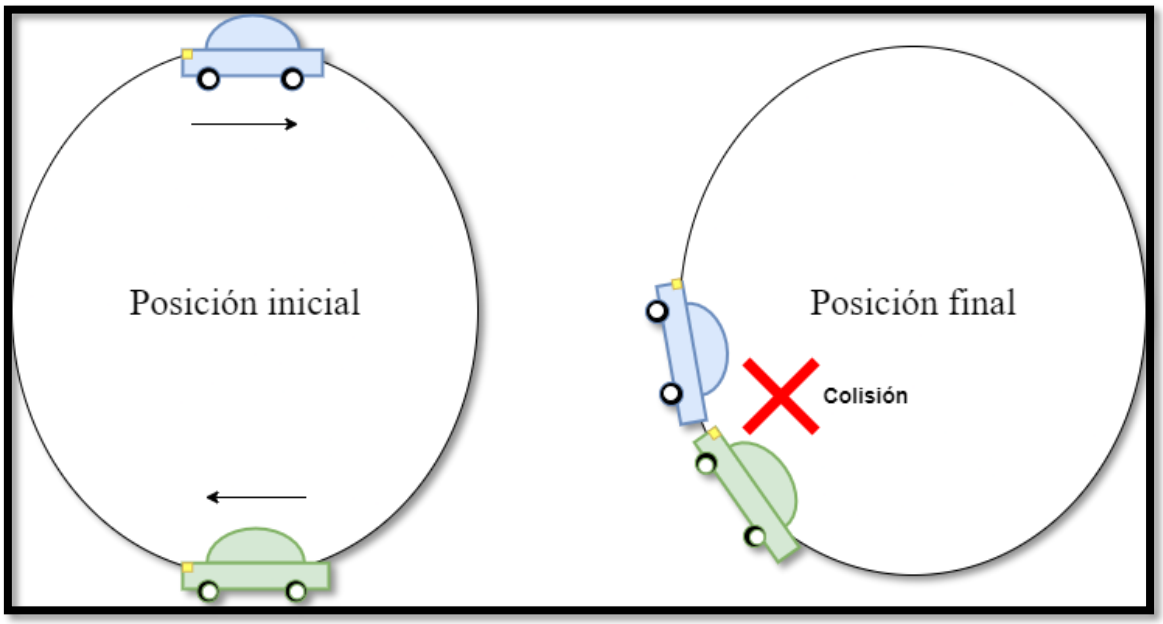

Figura 73.Comportamiento erróneo en el sistema con código SEC-DED Hamming (8,4).

Prueba 2: https://youtu.be/TFcBIAZURxE

- Sistema con código SEC-DAEC-TAED (8,3)

El tercer y último caso de estudio responde a la implementación del código SEC-DAEC-TAED desarrollado por metodologías propias. Este tiene la capacidad de corregir errores simples y dobles adyacentes además de detectar los triples adyacentes.

La inyección de fallos de estos tres tipos mencionados previamente es tolerada, y se traduce en el comportamiento del sistema esperado y, por tanto, no se produce ninguna colisión entre los vehículos.

En el caso de inyectar errores cuádruples adyacentes el comportamiento del sistema es erróneo y las instrucciones a realizar no se procesan correctamente. Concretamente, el vehículo seguidor no interpreta adecuadamente ningún comando, no realiza ninguna acción y provoca que finalmente el vehículo director lo alcance y provoque una colisión.

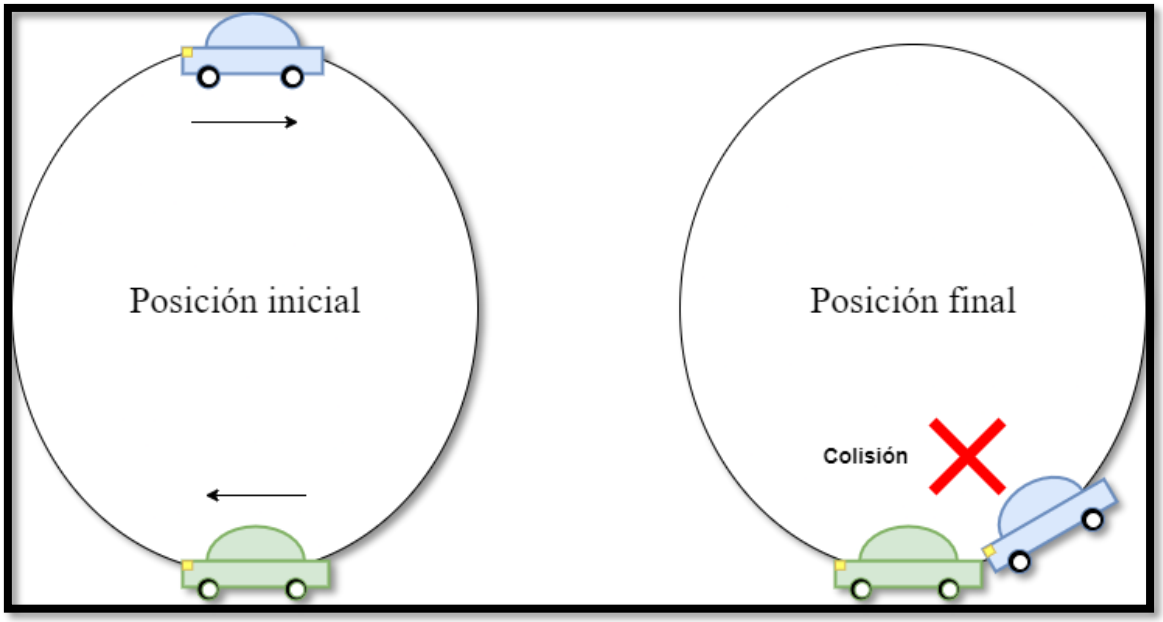

Figura 74. Comportamiento erróneo en el sistema con código SEC-DAEC-TAED (8,3).

Prueba 4: https://youtu.be/WN_ADNgpdG4

6.2 Evaluación del sistema

Una vez implementados los diferentes ECCs en los vehículos autónomos, junto con el circuito propuesto, y haber inyectado fallos de cuatro tipos en cada caso, se procede ahora a valorar los resultados obtenidos. Se va a cuantificar y comparar el tamaño del software requerido y los tiempos de ejecución para cada versión, es decir, una primera versión en la que la codificación se hace en la programación de cada vehículo y otra en la que las codificaciones de cada dato vienen dadas.

- Análisis del tamaño del software

	Vehículo Director				Vehículo seguidor			
	Sin codificador		Con codificador		Sin codificador		Con codificador	
ECC Implementado	**Software**	**Datos**	**Software**	**Datos**	**Software**	**Datos**	**Software**	**Datos**
Sin ECC	8870	647			6270	354		
Hamming(8,4)	9112	723	11314	1262	8220	1100	8476	1100
Hsiao(8,4)	9152	763	10146	1014	9660	920	9864	906
SEC-DAEC-TAED(8,3)	9154	765	10128	1007	9522	919	9664	905

Figura 75. Tamaño de software para cada vehículo y cada versión.

En la tabla anterior se muestra el tamaño en bytes de los programas y los datos asociados para ambos vehículos, aplicando distintos códigos, y para las versiones con codificadores y sin ellos. En la primera fila se observa la implementación sin códigos detectores de errores y por tanto no incluye versión con codificador.

En líneas generales, el software implementado con la versión en la cual se trabaja sin codificador ocupa menos espacio que la versión que sí lo implementa. Es un resultado más que esperado, debido a que, al añadir los métodos necesarios para la codificación en ambos vehículos, el software aumenta de tamaño.

Particularizando en este análisis, los diferentes ECCs implementados en el vehículo director presentan un tamaño muy parecido, tanto en el tamaño del software como en los datos necesarios para su correcta ejecución, debido a la poca diferencia entre las variables utilizadas en cada uno de ellos. Las diferencias surgen al añadir el codificador. En concreto, el código SEC-DAEC-TAED es el que menos tamaño de software y datos ocupa aun presentando la mayor capacidad de corrección de errores (aunque para menos bits de datos).

Con respecto al vehículo seguidor, los resultados son más variados que en el vehículo director. Esto se debe a que este vehículo (en ambas versiones, con y sin codificador) tiene que decodificar más datos codificados que el vehículo director. En concreto, la implementación del código SEC-DAEC-TAED es el que menor uso de datos utiliza en ambas versiones mientras que en el código Hamming el uso de datos es el mismo para ambas versiones. En general este uso de las variables necesarias es menor cuando se implementa la versión sin codificador, exceptuando al código Hamming en el que el uso es el mismo.

	Vehículo Director				Vehículo seguidor			
	Sin codificador		Con codificador		Sin codificador		Con codificador	
ECC Implementado	**Software**	**Datos**	**Software**	**Datos**	**Software**	**Datos**	**Software**	**Datos**
Sin ECC	27	31			19	17		
Hamming(8,4)	28	35	35	61	25	53	26	53
Hsiao(8,4)	28	37	31	49	29	44	30	44
SEC-DAEC-TAED(8,3)	28	37	31	49	29	44	29	44

Figura 76. Porcentaje de tamaño de software respecto a la memoria total.

Con el objetivo de ver el análisis de forma porcentual se obtiene la tabla anterior en la que se observan los porcentajes del tamaño del software y del uso de datos respecto al total de memoria disponible para software y para datos para cada ECC implementado y su versión.

En el caso del vehículo director se puede apreciar como los porcentajes son muy similares cuando se trabaja con la versión sin codificador. Sin embargo, cuando se trabaja con la versión con codificador el código Hamming es el ECC con los porcentajes más altos en tamaño de software y utilización de los datos.

Lo mismo ocurre con el vehículo seguidor. En este caso es también el código Hamming el que difiere respecto a los porcentajes del resto de implementaciones, siendo este el que menos porcentaje de tamaño de software requiere en ambas versiones, pero del que más porcentaje de uso de datos se ha obtenido, también en ambas versiones.

En general, el tamaño del software en ambos vehículos no presenta ningún problema de espacio, lo que permitiría añadir más funcionalidades extra para trabajos futuros.

- Análisis del tiempo de ejecución

	Vehículo Director		Vehículo seguidor	
ECC Implementado	**Sin codificador**	**Con codificador**	**Sin codificador**	**Con codificador**
Sin ECC	12		32	
Hamming(8,4)	14	23	180	183
Hsiao(8,4)	12	22	144	150
SEC-DAEC-TAED(8,3)	14	23	153	156

Figura 77. Tiempo de ejecución de cada vehículo para cada versión.

La tabla anterior muestra los tiempos de ejecución, expresados en milisegundos, de cada vehículo con las implementaciones de cada ECC y la versión utilizada en cada caso.

Lo primero que se aprecia es la gran diferencia de tiempos de ejecución que existe entre ambos vehículos. Esto se debe principalmente a dos causas. La primera es debido al tiempo que el vehículo seguidor debe esperar para la recepción de la palabra codificada por parte del vehículo director. La segunda razón es debida al tiempo de procesamiento de las palabras codificada por parte del segundo vehículo. Hay que recordar que el sentido de la comunicación es principalmente del vehículo director al vehículo seguidor. Únicamente cuando el segundo vehículo seguidor detecta algún fallo (el cual no puede corregir), el primer vehículo director ejecuta el decodificador para comprobar que la petición de reenvío mediante la variable comando es correcta.

Por otro lado, la diferencia de milisegundos entre las versiones con codificador y sin codificador son muy pequeñas en ambos vehículos, suponiendo una ventaja respecto a las

futuras implementaciones debido a la similitud de implementación entre las versiones con codificador y sin codificador. Sin embargo, estas diferencias pueden ser más notorias con la implementación de otros ECCs, a los cuales habría que volver a analizar sus resultados de tiempo de ejecución.

En definitiva, y observando el tiempo de ejecución en todos los casos dados, esto permite utilizar los ECC en este tipo de comunicaciones sin que el sistema pierda funcionalidades ni se retrase la respuesta de ambos vehículos.

7. Conclusiones

7.1 Conclusiones técnicas

Con el creciente aumento de la complejidad de los sistemas informáticos, el tratamiento de los datos es mayor y por tanto la posibilidad de errores en bits perdidos o invertidos en ellos se dispara. Por eso es necesario la aplicación de técnicas de corrección de errores como los ECCs, que además aumentan la confiabilidad al reducir la posibilidad de errores y garantizar una precisión en los datos enviados o almacenados. El ahorro de tiempo y recursos llega a ser un factor crítico en muchos sistemas, y gracias a estos códigos correctores de errores se permite aumentar la confiabilidad con incrementos mínimos del tiempo de proceso y de los recursos requeridos.

Durante la realización del proyecto ha sido posible el estudio detallado de la protección de las comunicaciones entre los vehículos seguidor y director, así como el almacenamiento de los datos en memoria, mediante la implementación de distintos ECCs. Dicha comprobación ha resultado en una protección totalmente viable y válida para la capacidad de corrección y detección de cada uno de los códigos correctores.

La técnica de inyección de fallos ha permitido evaluar el comportamiento de cada sistema con cada tipo de fallos. De esta forma, se han analizado las vulnerabilidades y fortalezas de cada ECC. En cambio, en un sistema real no controlado, los fallos pueden producirse de manera totalmente aleatoria. Pero si el comportamiento de los fallos reales se ajusta a las hipótesis de fallo con las que se ha diseñado cada código, la probabilidad de que se produzca una avería es muy baja.

Pese a que estos ECCs presentan una tolerancia a fallos simple, son un excelente punto de referencia o base de cara a los futuros estudios sobre ECCs más complejos y con mayor capacidad de almacenamiento de datos o cantidad de datos a transmitir por los canales de comunicación.

Todos ellos suponen un tamaño de software y un uso de los datos muy similares en las versiones con y sin codificador, y apenas ligeramente superior que la versión original, sin ECC. Lo mismo ocurre con los tiempos de ejecución de cada vehículo, los cuales no distan mucho entre cada versión. Por tanto, se pueden añadir nuevas funcionalidades y evitar retrasos en la transmisión de datos entre los vehículos.

Se ha comprobado además lo que supone para el comportamiento del vehículo seguidor una tolerancia a fallos nula. Mediante la inyección de fallos el sistema no es capaz de corregirlos ni tan solo detectarlos, provocando actuaciones erróneas o distintas a las requeridas.

En definitiva, la implementación de códigos correctores de errores debe depender de factores como la fiabilidad del canal de comunicaciones, la importancia de trabajar con datos críticos que puedan suponer errores graves o la sobrecarga que implica añadir ECCs en los códigos fuente. La necesidad de un ECC debe evaluarse en función de los requisitos y características de cada sistema o aplicación.

7.2 Relación con los estudios de máster

Durante el transcurso de los estudios del máster con nombre "Máster en Ingeniería de Computadores y Redes" han sido muchas las asignaturas cursadas las cuales han brindado de conocimientos y habilidades técnicas para llevar a cabo con éxito el desarrollo de este proyecto. Todo ello ha proporcionado una base sólida para abordar todos los desafíos y exigencias de manera efectiva.

En la asignatura de Sistemas Distribuidos y Empotrados (SED) se ha abordado un temario relacionado con el mundo del Internet de las Cosas (IoT). En primer lugar, se abordó de forma teórica el microcontrolador Arduino como ejemplo de circuito integrado al que se le puede grabar información mediante el entorno de desarrollo Arduino IDE. En este ámbito se vieron todas las ventajas que ofrece esta placa en relación con el mundo IoT. Mediante las clases prácticas se obtuvo una placa Arduino del tipo Uno junto con un kit en el que se podía trabajar con muchos módulos para aplicar funcionalidades muy diversas. El módulo Bluetooth HC-05 fue uno de ellos y en los que más hincapié se hizo durante el curso, aprendiendo todo sobre su conexión con la placa, la conectividad con otros módulos en modo esclavo y maestro, la capacidad que tiene para ser combinado con otros módulos actuadores e incluso la conexión a *routers* dentro de la propia aula. El aprendizaje de todo ello junto a los ejercicios prácticos planteados durante las clases supuso una habilidad técnica nueva y con la cual se han podido realizar las conexiones Bluetooth entre los vehículos seguidor y director sin mayor problema.

En la asignatura de Adaptabilidad y Reconfiguración en Soluciones Tolerantes a Fallos se han estudiado durante su transcurso todas las causas que provocan errores en los sistemas informáticos, así como los diferentes tipos de fallos que pueden llegar a ocurrir según que circunstancia. En el último bloque de la asignatura se explica el concepto de ECC y como estos funcionan para aumentar la tolerancia a posibles fallos. En dicho bloque se explica la base de estos códigos correctores de errores la cual comprende la adición de nueva información a los datos con los que se trabajan tanto en las comunicaciones como en el almacenamiento. Esta nueva información es añadida a los datos mediante procesos de codificación que dan lugar a una palabra codificada la cual deberá ser decodificado en el dispositivo receptor para extraer los datos enviados por el emisor. Se hace hincapié en los problemas que puede llegar a suponer la no detección de errores causados por fallos del sistema, así como los distintos errores a nivel de software existentes como los inyectados en este proyecto (adyacentes o simples). El código de

Hamming (7,4) sirve como ejemplo de la utilización de bits de paridad adicionales para la corrección de errores simples, así como la obtención de las matrices de paridad asociadas a cada ECC.

Estas dos asignaturas han sido las principales para obtener toda la información y conocimiento respecto a los sistemas empotrados y la tolerancia a fallos. De ese modo han sido posibles las tareas de configuración y conexión entre los módulos Bluetooth HC-05 así como la implementación de cada ECC propuesto en el proyecto.

7.3 Trabajos futuros

El proyecto final es una base sobre el estudio de la protección de las comunicaciones entre dos vehículos autónomos implementando una serie de códigos detectores de errores para comprobar la tolerancia a fallos, aplicando técnicas de inyección de fallos. Si bien esta tolerancia es más bien simple, es una buena forma de comenzar con sistemas más complejos.

- Transmisión de varias palabras de ocho bits

La librería de *SofwareSerial* permite la comunicación en serie mediante el uso de pines de la placa Arduino y utilizar sus métodos para la comunicación serial Bluetooth entre varios dispositivos. El método *write* es el encargado de enviar los datos mediante el pin de trasmisión configurado. El tamaño con el que este trabaja es siempre de ocho bits. En el proyecto se envían las palabras codificadas las cuales tienen un total de ocho bits, incluyendo los cuatro o tres bits de datos y el resto de los bits son de paridad, dependiendo del ECC.

En sistemas más complejos a diseñar se puede decidir el envío de más de una palabra de ocho bits consecutivas en cada transmisión. Estas palabras podrían contener valores numéricos o alguna otra información redundante del sistema para ser utilizado posteriormente por los componentes del vehículo receptor en cada caso.

- Mayor cantidad de datos a transferir

Los dos códigos SEC-DED trabajan con cuatro bits de datos, lo que supone una cantidad total de 16 posibles instrucciones a codificar. En el proyecto únicamente se trabaja con cuatro datos que contienen instrucciones a realizar por el vehículo seguidor. En el caso del código SEC-DAEC-TAED la cantidad de bits se reduce a tres siendo posible codificar hasta 8 instrucciones, de las cuales únicamente se utilizan cuatro en el proyecto.

Por tanto, se podría plantear aumentar el número de instrucciones posibles a enviar para evaluar nuevas funcionalidades en los vehículos, así como en los componentes añadidos. Tener en cuenta que el tamaño de software aumentaría en las versiones sin codificador implementado debido al aumento de la tabla correspondiente a cada codificación por dato.

- Adición de nuevos actuadores y sensores

Las posibilidades de añadir nuevos componentes a los vehículos vienen limitadas por la cantidad de pines que cada placa tenga disponible. En cualquier se pueden añadir nuevos sensores o actuadores teniendo en cuenta sus respectivas configuraciones y conexiones con la placa Arduino.

Un ejemplo práctico sería añadir diodos emisores de luz que actuaran en función de los valores de un sensor de iluminación en caso de que tener un circuito sin posibilidad de obtener luz artificial. Otro caso sería el uso de una funcionalidad de alarma en caso de que el vehículo seguidor detectase un error que no pueda corregir para evaluar en un sistema aislado cuantas veces se ha emitido este sonido de alarma.

7.4 Valoración personal

Durante el curso académico han sido muchas las asignaturas que han causado gran interés en mí. La primera de ellas fue la relacionada con los sistemas empotrados en la que el mundo Arduino me fascinó por su fácil aprendizaje y sobre todo utilidad para el entorno doméstico. Poco después conocí la asignatura de Adaptabilidad y Reconfiguración en Soluciones Tolerantes a Fallos y a sus profesores a los cuales les pedí realizar el Trabajo Final de Máster.

Desde primera instancia el proyecto que me propusieron y sus objetivos me parecieron una muy buena propuesta por lo que decidí comenzar a implementarlo. A los vehículos dados se debía incorporar una serie de componentes para la creación de un canal de comunicación entre ellos para el envío y recepción de datos, así como incluir los códigos necesarios en el entorno de desarrollo para construir un sistema con ECCs.

Los códigos correctores de errores implementados en el sistema son totalmente eficaces contra los errores que estos pueden tolerar, destacando el código SEC-DAEC-TAED el cual fue programado mediante las metodologías propias y los conocimientos adquiridos durante el curso 2021-2022.

Adicionalmente, junto con la ayuda de los tutores, se ha escrito un artículo científico con nombre “Protección de comunicaciones entre vehículos autónomos mediante el uso de códigos de corrección de errores” con toda la información reflejada de este proyecto y

enviado a las Jornadas SARTECO (2023) el cual fue aprobado y por tanto aceptado, así que añado una publicación científica a mi expediente en la UPV y de la cual me siento bastante orgulloso.

En definitiva, se ha hecho un largo trabajo desde el comienzo en el que se debía corregir múltiples configuraciones en ambos vehículos hasta la programación con Arduino IDE para la implementación de los ECCs y cuyas recompensas han finalizado en unos escenarios con sistemas tolerantes a fallos y en los cuales se pueden añadir nuevas funcionalidades.

8. Referencias

8.1 Bibliografía

[1] J.Lázaro Trazón, “Diseño e implementación mediante aplicaciones CAD e impresión 3D de un coche a escala guiado por Arduino”, Trabajo Final de Grado, Universidad Politécnica de Valencia, 2021, http://hdl.handle.net/10251/170866

[2] R. García Armero, “Desarrollo de un prototipo a escala de un vehículo para pruebas de laboratorio mediante impresión 3D”, Trabajo Final de Grado, Universitat Politècnica de València, 2021, http://hdl.handle.net/10251/173301

[3] J. Gracia-Morán, “Protección de comunicaciones entre vehículos autónomos mediante el uso de códigos de corrección de errores”, Instituto ITACA,2023.

[4] Arduino, “SoftwareSerialLibrary”, ArduinoDocs, 2023, https://docs.arduino.cc/learn/built-in-libraries/software-serial#syntax

[5] R. Castellor, “Fiabilidad y tolerancia a fallos”, UNED, 2017, https://uned-sistemas-tiempo-real.readthedocs.io/es/latest/tema02.html

[6] N.Mechatronics, “CONFIGURACIÓN DEL MÓDULO BLUETOOTH HC-05 USANDO COMANDOS AT”, 2018, https://naylampmechatronics.com/blog/24_configuracion-del-modulo-bluetooth-hc-05-usando-comandos-at.html

[7] Y.Fernández, “Qué es Arduino, cómo funciona y qué puedes hacer con uno”, Xataca Basics, https://www.xataka.com/basics/que-arduino-como-funciona-que-puedes-hacer-uno

[8] L.del Valle Hernández, “Servomotor con Arduino tutorial de programación paso a paso”, Programarfacil.com, 2023, https://programarfacil.com/blog/arduino-blog/servomotor-con-arduino/

[9] G.Notte, “Clase Codigo De Hamming”, Youtube, 2018, https://youtu.be/zg06eShv6ok

[10] Cambatronics Online, “Comunicación Maestro/ esclavo con dos módulos bluetooth y arduino.”, Youtube, 2022, https://youtu.be/crxukoldLOs

[11] R.Invarato, “Código de Hamming: Detección y Corrección de errores”, Jarroba,2016,https://jarroba.com/codigo-de-hamming-deteccion-y-correccion-de-errores/

[12] R. Razavi; M. Fleury; M. Ghanbari, “Correct Bluetooth EDR FEC Performance with SEC-DAEC Decoding”, ResearhGate, 2016, https://www.researchgate.net/publication/3403969_Correct_bluetooth_EDR_FEC_performance_with_SEC-DAEC_decoding

[13] J. Gracia-Morán, A. Vicente-García y L.-J. Saiz-Adalid, "Protección de comunicaciones entre vehículos autónomos mediante el uso de códigos de corrección de errores", aceptado para publicación en Jornadas SARTECO, Ciudad Real, Septiembre 2023.