_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Vehículo autónomo multisensorizado para la identificación y seguimiento de caminos definidos por diferentes variables físicas

Juan Trinidad Jiménez Armesto

Tutor: Dr. D. Ángel Héctor García Miquel

Cotutor: Dr. D. Vicente Torres Carot

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2015-16

Valencia, 13 de Septiembre de 2016

A mi padre, ejemplo de trabajo y superación.

Resumen

En este Trabajo de Fin de Grado (TFG) se ha diseñado e implementado un vehículo multisensorizado que se desplaza de forma automática por un circuito creado para este proyecto. Se encuentra dotado de información sensorial de diferentes orígenes o principios físicos: ópticos, inductivos, térmicos, ultrasónicos, etc., permitiendo así el desplazamiento autónomo por el circuito en función de estos principios.

También se ha realizado un sistema de comunicación a través de Bluetooth con el coche mediante una aplicación móvil instalada en un dispositivo Android. Esta aplicación permite dos modos de funcionamiento en el vehículo: automático, en el que se basa el TFG, consistente en el movimiento autónomo del vehículo en función de los principios comentados anteriormente; y otro es el movimiento manual, que se ha desarrollado como complemento del anterior y permite el control total del vehículo.

El sistema está constantemente en comunicación mediante Bluetooth, lo que permite monitorizar e interaccionar con el vehículo, así como manejarlo en tiempo real o tener en la aplicación la información que nos envía el vehículo.

Todo esto se ha realizado con las herramientas *Arduino*, para programar el vehículo, y *Android Studio*, para programar la aplicación.

Abstract

On this End of Degree Project it has been designed and implemented a vehicle with numerous sensors the moves automatically on a circuit created for this project. The circuit has different origins or physical principles of sensorial information: optical, inductive, thermal, ultrasonic, etc. so allowing the autonomous movement on the circuit according to those principles.

It has also been designed a mobile application for an Android device for a Bluetooth communication with the car. This application allows two types of operation: automatic movement, in which this project is based, consisting of an automatic movement according to the principles discussed above in mind; and a manual movement; it has been developed as an accessory and permits the total control of the car.

The system is in constant communication through Bluetooth, this allows you monitor the car and interact with it and move it in real time or have all the information that the car sends to us on our application as well.

All this has been done with Arduino tools, to program the vehicle, and Android Studio, to program the application.

Resum

En este Treball de Fi de Grau (TFG) s'ha dissenyat i implementat un vehicle multisensoritzat que es desplaça de forma automática per uncircuit creat per al projecte. Este està dotat d'imformació sensorial de diferents orígens o principis físics: óptics, inductius, tèrmics, ultrasónics, etc., permetent així el desplaçament autónomament per el circuit en funció d'estes variables.

També s'ha realitzat un sistema de comunicació a través de Bluetooth per mitjà d'una aplicació mòbil instal·lada en un dispositiu Android. Esta aplicació permet dos modes de funcionament en el vehicle: automàtic, en el que es basa el TFG, consistent en el moviment autònom del vehicle en funció de les variables físiques comentades anteriorment; i un altre és el moviment manual, que s'ha desenvolupat com a complement de l'anterior i permet el control total del vehicle

El sistema està constantment en comunicació per mitjà de Bluetooth, la qual cosa permet monitoritzar i interaccionar amb el vehicle, així com manejar-ho en temps real o tindre en l'aplicació la informació que ens envia el vehicle.

Tot açò s'ha realitzat amb les ferramentes Arduino, per a programar el vehicle, i Android Studio, per a programar l'aplicació.

Índice

Capítulo 1. Introducción, objetivos y metodología

Un vehículo autónomo, también conocido como robótico, o, informalmente, vehículo sin conductor, es un automóvil capaz de imitar las capacidades humanas de manejo y control del mismo, pudiendo percibir el medio que le rodea y navegar en consecuencia. El conductor podrá elegir el destino, pero no se requiere su interacción para activar ninguna operación mecánica del vehículo.

Los vehículos autónomos reciben información del entorno mediante técnicas complejas como láser, radar, LIDAR, sistema de posicionamiento global, visión computarizada, etc. Los sistemas avanzados de control interpretan la información para identificar la ruta de destino más apropiada, así como los obstáculos y la señalización relevante. Los vehículos autónomos generalmente son capaces de circular por carreteras previamente programadas y requieren una reproducción cartográfica del terreno. Por tanto, si una ruta no se encuentra incluida en el sistema, puede darse el caso de que el vehículo no pueda avanzar de forma coherente y normal.

Este Trabajo Fin de Grado (TFG) persigue dos metas ambiciosas. Por un lado, el diseño y la construcción de un vehículo multisensorizado capaz de desplazarse de forma autónoma siguiendo diferentes trayectorias en función de la luz (generada por un diodo), el color (de cintas adhesivas y diodos led de un “semáforo”) y las distancias (respecto a diversos obstáculos). Y, por otro, la creación de una aplicación móvil desde la cual manejar el vehículo y recibir información en tiempo real de su situación (estado, acción actual, etc.).

En el montaje tanto vehículo como del "semáforo" nos apoyaremos en las placas Arduino, a las que conectaremos diversos sensores y otros componentes electrónicos, como motores, zumbadores, etc. La aplicación móvil será programada en Android y comunicada con el vehículo mediante Bluetooth.

El movimiento del vehículo dispondrá de dos modos de funcionamiento, uno bajo control manual y otro automático, ambos conexionados al dispositivo móvil mediante Bluetooth, como se acaba de indicar.

El control manual consiste en el manejo total del vehículo por parte del usuario, pudiendo moverlo en ocho direcciones diferentes, además de pararlo, encender los faros, hacer que suene el claxon, controlar la velocidad y realizar medidas de temperatura, humedad y distancia hasta el objeto más cercano por la parte frontal del vehículo.

El control automático consistirá en el movimiento autónomo por parte del vehículo en función del entorno en el que se encuentre. Se diseñó un circuito para el movimiento del vehículo formado por tres “entornos”. El primero consiste en el movimiento del vehículo evitando una línea negra situada en el suelo, lo que hará que realice el camino marcado por ésta. Se desplazará de esa forma hasta que se encuentre con una franja negra perpendicular a la línea, lo que hará que se cambie a un segundo "entorno" en donde, además de seguir evitando las líneas negras, cada vez que vuelva a encontrar una franja negra perpendicular se detendrá para medir la distancia frontal por si está el semáforo y en caso contrario habrá un cuadrado de color en el suelo

para que el vehículo obtenga el color que es. Por último, llegará al "semáforo", en el cual esperará a que éste pase de rojo a verde para cambiar al último "entorno", que consistirá en un movimiento libre del vehículo evitando obstáculos.

Estos dos controles del vehículo se elegirán desde la aplicación, en la cual recibiremos y mostraremos información en tiempo real, tanto de lo que está realizando el vehículo cuando se trate del movimiento automático, como de las mediciones de los sensores en ambos movimientos. Además de controlar el vehículo, la aplicación dispone de un apartado donde observar la información sobre los componentes utilizados, y otro en el que se describe el proyecto.

Planteados estos objetivos, el trabajo se dividirá en dos grandes bloques: el correspondiente al hardware utilizado tanto en el vehículo como en el "semáforo" (capítulo 2), y el del software utilizado para programar, por un lado, el vehículo y el "semáforo" en las placas Arduino, y por otro, la aplicación Android (capítulo 3); pudiendo ver los resultados obtenidos en el capítulo 4. A estos dos bloques se acompañan otros capítulos orientados, uno, a mostrar los costes de fabricación del producto, su precio de venta y el umbral de rentabilidad o punto muerto que nos indique el número de vehículos que hemos de producir para empezar a tener beneficios (capítulo 5), y otro a estudiar la posible implementación real de este vehículo en una empresa (capítulo 6), finalizando con el capítulo dedicado a las conclusiones y líneas futuras de continuación del proyecto, la bibliografía y los anexos.

Cronológicamente, para el desarrollo de estos apartados se han ido realizando las siguientes tareas, desarrolladas en los capítulos que siguen a continuación:

1. Estudio teórico de los sensores, programación individual en Arduino y estudio de los resultados.
2. Montaje del vehículo con cada uno de los sensores de forma independiente para, una vez vistos los resultados individuales, unirlos todos y contrastar el resultado conjunto.
3. Organización del código de programación del vehículo en dos partes (control manual y control automático).
4. Programación de la aplicación móvil.
5. Comunicación entre el vehículo y la aplicación mediante Bluetooth.
6. Estudio del comportamiento del vehículo usando la aplicación.

Capítulo 2. Implementación hardware

Los elementos hardware de nuestro proyecto los hemos clasificado en dos partes: el vehículo y un "semáforo". El vehículo consta de un chasis sobre el que se instalará el controlador, los sensores y los motores necesarios para su funcionamiento, además de los diferentes componentes para las conexiones, alimentación, etc. El "semáforo" estará formado por otro controlador, un único sensor y componentes luminosos. Ambos controladores serán placas de Arduino.

2.1 Arduino

Arduino es una plataforma de creación de prototipos de código abierto basada en el uso sencillo de hardware y software. Se basa en placas de desarrollo que integran un microcontrolador y un entorno de desarrollo (IDE), diseñado para facilitar el uso de la electrónica en proyectos multidisciplinarios. Estas placas están diseñadas para que sean capaces de leer entradas (luz en un sensor, activación de un botón, etc.) y transformarlas en una salida (activar un motor, encender un led, etc.).

El hardware consiste en una placa de circuito impreso con un microcontrolador, usualmente Atmel AVR, y puertos digitales y analógicos de entrada/salida, los cuales pueden conectarse a placas de expansión (shields) que amplían sus características de funcionamiento.

Su software consiste en un entorno de desarrollo (IDE) basado en el entorno de Processing y lenguaje de programación basado en Wiring, así como en el cargador de arranque (bootloader) que es ejecutado en la placa. El microcontrolador de la placa se programa a través de un computador, haciendo uso de la comunicación serial mediante un convertidor de niveles RS-232 a TTL serial.

Existe una gran variedad de placas Arduino como el Arduino UNO, Nano, Mega, Leonardo, etc., y para este proyecto se han utilizado el Arduino UNO y el Arduino Mega. Obviamente, será el corazón del proyecto, ya que será donde se programen las instrucciones de las operaciones a realizar.

2.1.1 Arduino UNO

El Arduino UNO será el controlador de nuestro "semáforo". Esta placa electrónica basada en el microcontrolador ATmega328 contiene todo lo necesario para su funcionamiento, bastando con conectarla al ordenador mediante un cable USB o alimentarlo con una batería externa.

En este proyecto vamos a utilizar la nueva versión del Arduino Uno, el Arduino Uno R3 (figura 2.1). El Arduino Uno R3 utiliza el microcontrolador ATmega16U2 para el manejo de USB en lugar del 8U2, lo que permite ratios de transferencia más rápidos y más memoria. No se necesitan drivers para Linux o Mac (el archivo *.inf* para Windows es necesario y está incluido en el IDE de Arduino).

La tarjeta Arduino Uno R3 incorpora pines SDA (línea por donde se envían datos) y SCL (línea por donde se sincroniza la transferencia de datos). También incluye el pin IOREF, que permite a los shields adaptarse al voltaje brindado por la tarjeta, y otro pin sin conectar para futuros propósitos. La tarjeta trabaja con todos los shields existentes y podrá adaptarse con los nuevos shields utilizando esos pines adicionales.

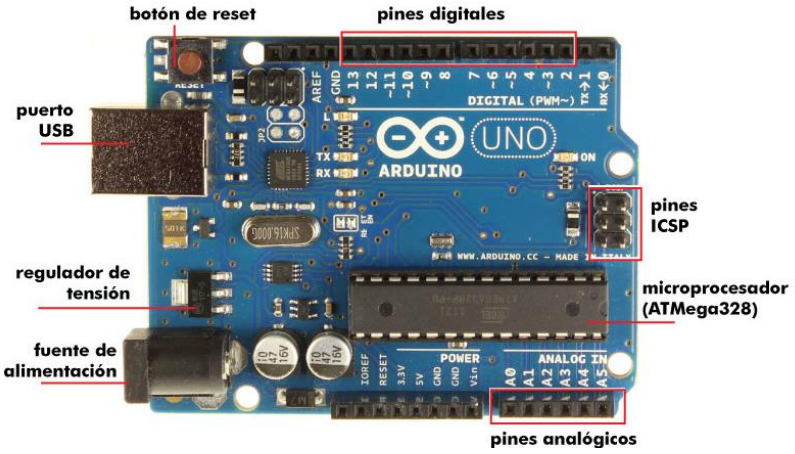

Figura 2.1: Arduino UNO R3

Especificaciones técnicas

- Microprocesador ATmega328.
- Voltaje de operación: 5V.
- Tensión de trabajo: 7-12V.
- Corriente continua por pines E/S: 40mA.
- Corriente continua del pin de alimentación 3,3V: 50mA.
- 14 pines digitales E/S (de los cuales 6 pueden ser utilizados como salidas PWM).
- 6 pines analógicos E/S.
- Frecuencia de reloj (oscilador de cristal): 16MHz.
- Conexión USB 2.0 tipo estándar.
- Cabecera de ICSP.
- Botón de reinicio.
- Memoria flash de 32kB (ATmega328) de los cuales 0,5kB se usan para el arranque.
- 2kB de memoria SRAM (ATmega328).
- 1kB de memoria EEPROM (ATmega328).

Alimentación

El Arduino UNO puede ser alimentado vía conexión USB o con una fuente de alimentación externa. El origen de la alimentación se selecciona automáticamente.

Las fuentes de alimentación externas (no-USB) pueden ser tanto un transformador o una batería. El transformador se puede conectar usando un conector macho de 2,1mm con centro positivo en el conector hembra de la placa. En caso de usar batería, sus cables pueden conectarse a los pines GND y V_{IN} en los conectores de alimentación (POWER).

La placa puede trabajar con una alimentación externa de entre 6 y 20 voltios, siendo el rango recomendado el comprendido entre 7 y 12 voltios, ya que si el voltaje suministrado es inferior a 7V, el pin de 5V puede proporcionar menos de 5 voltios y la placa puede volverse inestable; si se usan más de 12V, los reguladores de voltaje se pueden sobrecalentar y dañar la placa.

Los pines que posee la placa para suministrar alimentación son los siguientes:

- VIN: es la entrada de voltaje a la placa Arduino cuando se está usando una fuente externa de alimentación. Se puede proporcionar voltaje a través de este pin, o, si se está alimentando a través de la conexión de 2,1mm, acceder a ella a través de este pin.
- 5V: es una fuente de voltaje estabilizado usado para alimentar el microcontrolador y otros componentes de la placa. Ésta puede provenir de V_{IN}, a través de un regulador integrado en la placa, o proporcionada directamente por el USB u otra fuente estabilizada de 5V.
- 3V3: es una fuente de voltaje de 3,3 voltios generada por un regulador integrado en la placa. La corriente máxima soportada es de 50mA.
- GND: son pines de toma de tierra.

Memoria

El ATmega328 tiene 32kB de memoria flash para almacenar código, de los que 0,5kB son usados para el arranque del sistema (bootloader). Tiene además 2kB de memoria SRAM y 1kB de EEPROM, a la cual se puede acceder para leer o escribir con la librería EEPROM.

Entradas y Salidas

Cada uno de los 14 pines digitales que posee el UNO pueden utilizarse como entradas o como salidas usando las funciones pinMode(), digitalWrite(), y digitalRead(). Las E/S operan a 5 voltios. Cada pin posee una resistencia interna de 20kΩ (desconectada por defecto) a la que se accede mediante software y que se utilizan para conectar las entradas a interruptores. Cada pin configurado como "salida" puede proporcionar una intensidad máxima de 40mA. Los 6 pines analógicos también pueden utilizarse como entradas y salidas y proporcionan una resolución de 10bits (1024 valores). Por defecto se mide desde 0V a 5V, aunque es posible cambiar la cota superior de este rango usando el pin AREF y la función analogReference(), ya que este pin configura la referencia de tensión utilizada para la entrada analógica.

Además, algunos pines tienen funciones específicas:

- Comunicación serie: pines 0 (RX) y 1 (TX). Usados para recibir (RX) y/o transmitir (TX) datos a través de puerto serie TTL. Estos pines se conectan a los correspondientes del chip FTDI USB-to-TTL.

- Interrupciones externas: pines 2 (interrupción 0) y 3 (interrupción 1). Estos pines se pueden configurar para lanzar una interrupción en un valor LOW (0V), en flancos de subida o bajada (cambio de LOW a HIGH (5V) o viceversa), o en cambios de valor.
- PWM: pines 3, 5, 6, 10, 11. Proporcionan una salida PWM (Pulse Wave Modulation, modulación de onda por pulsos) de 8 bits de resolución (valores de 0 a 255) a través de la función analogWrite().
- SPI: pines 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK). Estos pines proporcionan comunicación mediante el protocolo de comunicación síncrona SPI usando la librería SPI.
- LED: pin 13. Hay un LED integrado en la placa conectado al pin digital 13; cuando este pin tiene un valor HIGH (5V) el LED se enciende, y cuando tiene un valor LOW (0V), se apaga.
- I2C: pines A4 (SDA) y A5 (SCL). Sirven de soporte para el protocolo de comunicaciones I2C (TWI) usando la librería Wire.
- AREF. Voltaje de referencia para las entradas analógicas. Usado por analogReference().
- Reset. Suministra un valor LOW (0V) para reiniciar el microcontrolador. Normalmente es usado para añadir un botón de reset a los shields que no dejan acceso a este botón en la placa.

2.1.2 Arduino Mega

Arduino Mega 2560 será el controlador de nuestro vehículo (figura 2.2). Es un modelo más potente que el anterior y está basado en el microcontrolador ATmega2560.

Una de las características principales de esta tarjeta es que no utiliza el convertidor USB-serie de la firma FTDI, sino que emplea un microcontrolador ATmega8U2 programado como actual convertidor USB a serie.

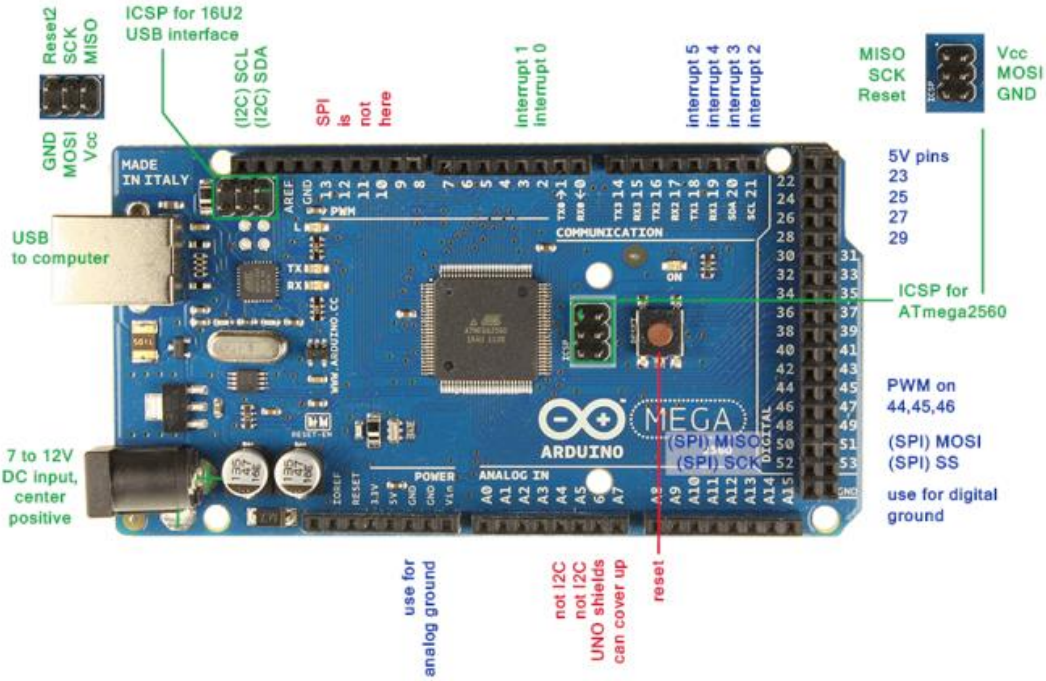

Figura 2.2: Arduino Mega

Especificaciones técnicas

- Microprocesador ATmega2560.
- Voltaje de operación: 5V.
- Tensión de trabajo: 7-12V.
- Integra regulación y estabilización de +5Vcc.
- Corriente continua por los pines E/S: 40mA.
- Corriente continua del pin de alimentación 3.3V: 50mA.
- 54 pines digitales E/S (14 de ellos se pueden utilizar como salidas PWM).
- 16 pines analógicos E/S.
- Frecuencia de reloj: 16MHz.
- Conexión USB 2.0 tipo estándar.
- Cabecera de ICSP.
- Botón de reinicio.
- Memoria flash de 256kB (ATmega2560), de los cuales el bootloader ocupa 8kB.
- 8kB de memoria SRAM.
- 4kB de memoria EEPROM.
- 4 UARTs (puerto serie).

Alimentación

Posee las mismas características que las especificadas en el apartado de Alimentación del Arduino UNO.

Memoria

El microprocesador ATmega2560 tiene 256kB de memoria flash para almacenar código, de los que 8kB son usados para el arranque del sistema (bootloader). Tiene 8 kB de memoria SRAM y 4kB de memoria EEPROM, a la cual se puede acceder para leer o escribir a través de la librería EEPROM.

Entradas y Salidas

Cada uno de los 54 pines digitales contenidos en el Mega pueden utilizarse como entradas o como salidas usando las funciones pinMode(), digitalWrite() y digitalRead(). Las E/S operan a 5 voltios. Cada pin posee una resistencia interna de 20kΩ (desconectada por defecto) a la que se accede mediante software y que se utilizan para conectar las entradas a interruptores. Cada pin configurado como "salida" puede proporcionar una intensidad máxima de 40mA. Los 16 pines analógicos también permiten ser configurados como entradas y salidas y cada una de ellas proporciona una resolución de 10bits (1024 valores). Por defecto se mide desde 0V a 5V, aunque es posible cambiar la cota superior de este rango usando el pin AREF y la función analogReference().

Como en el caso anterior, algunos pines tienen funciones específicas:

- Comunicación serie: pines 0 (RX) y 1 (TX), comunicación serie 1: pines 19 (RX) y 18 (TX); comunicación serie 2: pines 17 (RX) y 16 (TX); comunicación serie 3: pines 15 (RX) y 14 (TX). Usados para recibir (RX) y/o transmitir (TX) datos a través del puerto

serie TTL. Los pines serie: 0 (RX) y 1 (TX) están conectados a los pines correspondientes del chip FTDI USB-to-TTL.

- Interrupciones externas: pines 2 (interrupción 0), 3 (interrupción 1), 18 (interrupción 5), 19 (interrupción 4), 20 (interrupción 3) y 21 (interrupción 2). Estos pines se pueden configurar para lanzar una interrupción en un valor LOW (0V), en flancos de subida o bajada (cambio de LOW a HIGH (5V) o viceversa), o en cambios de valor.
- PWM: pines 2 a 13. Proporciona una salida PWM (Pulse Wave Modulation, modulación de onda por pulsos) de 8 bits de resolución (valores de 0 a 255) a través de la función analogWrite().
- SPI: pines 53 (SS), 51 (MOSI), 50 (MISO) y 52 (SCK). Estos pines proporcionan comunicación SPI, usando la librería SPI.
- LED: pin 13.
- I2C: pines 20 (SDA) y 21 (SCL). Soporte para el protocolo de comunicaciones I2C (TWI) usando la librería Wire.
- AREF.
- Reset.

2.1.3 Entorno de desarrollo

Un entorno de desarrollo integrado, llamado también IDE (en inglés, Integrated Development Environment), es un programa informático compuesto por un conjunto de herramientas de programación. Puede dedicarse en exclusiva a un lenguaje de programación o bien utilizarse para varios lenguajes.

El entorno de desarrollo de Arduino (figura 2.3) está compuesto por un editor de texto, una consola, un área de mensajes y dos barras de herramientas para los botones de acceso rápido (verificar, subir, nuevo, abrir, guardar y monitor serie) y menús.

Los programas generados con el software Arduino se denominan *sketches* y se basan en el lenguaje C. Éstos se escriben sobre el área blanca denominada “edición de texto”. Debajo de ella se encuentra la “zona de mensajes” donde se despliegan los posibles errores de código, cuya descripción viene en el campo “consola”.

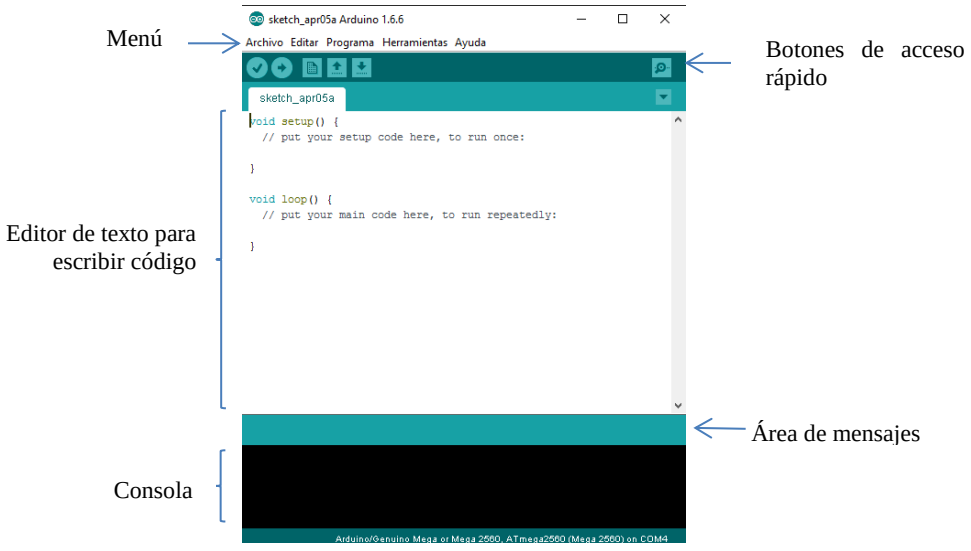

Figura 2.3: Entorno de desarrollo Arduino

2.2 Motores servo

Un servomotor, o también llamado servo, es un motor similar a un motor de corriente continua pero tiene la capacidad de ubicarse en cualquier posición dentro de un rango de operación y mantenerse estable en dicha posición. Este tipo de motor eléctrico es capaz de ser controlado tanto en velocidad como en posición mediante variaciones de pulsos.

En nuestro proyecto, los motores serán los encargados de mover tanto el vehículo como un sensor de ultrasonidos. La velocidad del vehículo variará en función del tipo de movimiento y del entorno en el que se encuentre, ya que en el control manual podremos manejarlo con la velocidad que queramos, mientras que en el automático el vehículo irá más despacio para realizar las medidas con más exactitud. El servo que mueva el sensor de ultrasonidos realizará un movimiento lento para que el sensor mida tanto de frente como a los lados la distancia hasta los objetos más próximos al vehículo.

Características

- Consumo de energía reducido.
- Corriente requerida en función del tamaño (ésta depende principalmente del par).
- Giran en un rango de 180 grados. Sin embargo, los hay que permiten los 360 grados.
- Compuestos por un sistema de control (tarjeta electrónica), un potenciómetro y un conjunto de engranajes.
- 3 cables de conexión externa: rojo para alimentación (5V), negro o marrón para tierra (GND) y naranja o blanco para el control.

Funcionamiento

Los servomotores hacen uso de la modulación por ancho de pulso (PWM) para controlar la dirección o posición (figura 2.4). La mayoría trabajan en la frecuencia de los 50Hz, por lo que las señales PWM tienen un período de 20ms. La electrónica dentro del servomotor responderá al ancho de la señal modulada. Los valores más generales se corresponden con pulsos de entre 1ms y 2ms de anchura, que dejarían al motor en ambos extremos (0º y 180º), mientras que un pulso de 1,5ms representa un estado neutro (90º).

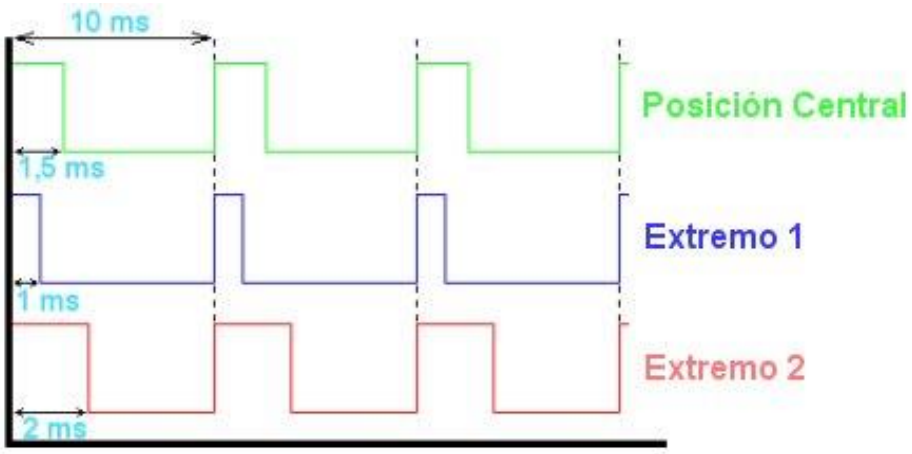

Figura 2.4: Tren de pulsos para controlar el servo

Conexionado

Cables negro y rojo a la alimentación (GND y 5V), y el cable amarillo al pin PWM configurado en el código como salida de señal para el motor (figura 2.5).

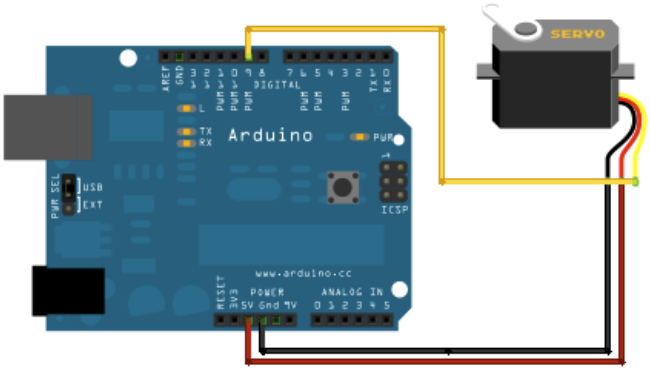

Figura 2.5: Conexión servo

Código

En este proyecto se ha decidido programar los motores mediante *timers*. Un *timer* o temporizador se utiliza para realizar acciones periódicas (comprobar el estado de un pulsador, o si se han recibido datos por el puerto serie, por ejemplo), calcular el tiempo transcurrido entre dos sucesos, generar señales, etc.

Un modo de funcionamiento de *timer* es *Clear Time on Compare* (CTC mode). Consiste en un contador que se actualiza con una frecuencia programable y que puede avisar cuando la cuenta alcanza un valor previamente grabado (que llamaremos CuentaMAX) en uno de los registros de control del temporizador. Cuando se alcanza el valor CuentaMAX, el temporizador se resetea y vuelve a contar desde 0. La frecuencia con la que el temporizador alcanzará dicho valor se obtiene mediante la ecuación 2.1.

$$f_{max} = \frac{1}{(CuentaMax+1)\frac{16MHz}{preescalado}} \qquad \text{Ecuación 2.1}$$

Nótese que la expresión anterior es válida para la placa Mega2560, cuyo reloj es de 16MHz.

El valor de preescalado puede ser 1, 8, 64, 256 ó 1024. El valor de CuentaMAX no puede exceder de 65535, máximo valor codificable con los 16 bits del temporizador. Los registros de control implicados son los siguientes:

- La lectura de TCNT3H y TCNT3L devuelve el estado actual de la cuenta.
- Los registros OCR3A o ICR3 pueden ser empleados para almacenar CuentaMAX.
- Dependiendo del registro empleado en la comparación, se activará el flag OCF3A o el ICF3. Si los bits OCE3A/ICE3 del registro TIMSK3 están activos, se producirá una interrupción.
- Los registros TCCR3A y TCCR3B permiten seleccionar el modo CTC.

Los registros deben configurarse según aparecen en la figura 2.6.

Bit	7	6	5	4	3	2	1	0	
(0xA0)	COM4A1	COM4A0	COM4B1	COM4B0	COM4C1	COM4C0	WGM41	WGM40	TCCR4A
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
(0xA1)	ICNC4	ICES4	–	WGM43	WGM42	CS42	CS41	CS40	TCCR4B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.6: Configuración de los registros

Los bits CS32, CS31 y CS30 del TCCR3B permiten fijar el preescalado del registro, como queda especificado en la figura 2.7.

Clock Select Bit Description

CSn2	CSn1	CSn0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on Tn pin. Clock on falling edge
1	1	1	External clock source on Tn pin. Clock on rising edge

Figura 2.7: Preescalado del registro

Los bits WGM31 y WGM30 de TCCRA y el WGM33 y WGM32 de TCCRB permiten seleccionar el modo CTC, siguiendo los criterios de la figura 2.8.

Mode	WGMn3	WGMn2 (CTCn)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Timer/Counter Mode of Operation	TOP	Update of OCRnx at	TOVn Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCRnA	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
9	1	0	0	1	PWM,Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICRn	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCRnA	TOP	BOTTOM
12	1	1	0	0	CTC	ICRn	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICRn	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCRnA	BOTTOM	TOP

Figura 2.8: Selección modo funcionamiento

Nótese que el bit WGM43 determina si se usa OCR3A o ICR3 para detectar el final de cuenta (TOP).

El registro TIMSK3 permite habilitar las interrupciones según se establece en la figura 2.9.

Bit	7	6	5	4	3	2	1	0	
(0x72)	–	–	ICIE4	–	OCIE4C	OCIE4B	OCIE4A	TOIE4	TIMSK4
Read/Write	R	R	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.9: Habilitación de las interrupciones

Cuando se alcance la cuenta máxima, los flags de interrupciones se activarán en el registro TIFR3, quedando como se muestra en la figura 2.10.

Bit	7	6	5	4	3	2	1	0	
0x19 (0x39)	–	–	ICF4	–	OCF4C	OCF4B	OCF4A	TOV4	TIFR4
Read/Write	R	R	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.10: Activación del registro TIFR3

A continuación presentaremos un ejemplo de código que hace moverse una de las ruedas del vehículo. Para ello emplea el temporizador 3, en el que se definen dos valores de comparación, de tal manera que cuando son alcanzados ambos valores, se producen sendas interrupciones. La primera interrupción pone la salida conectada al motor a 0, mientras que la segunda la pone a 1 y reinicia la cuenta.

El temporizador aumenta su cuenta con una frecuencia 16MHz/64, lo que da un periodo de 4µs. Si, por ejemplo, se quiere que la primera comparación se cumpla al cabo de 1,8 ms, el valor de la cuenta a emplear será CompA = 1,8 ms/4µs = 450. La segunda comparación debe fijarse para que se cumpla a los 20ms. Esto puede observarse en la figura 2.11.

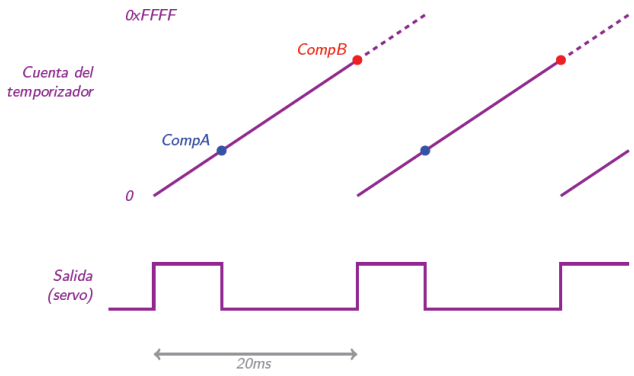

Figura 2.11: Funcionamiento temporizador

Código:

```
#include <Arduino.h>
#include <HardwareSerial.h>
#include <avr/interrupt.h>
#include <string.h>
extern HardwareSerial Serial;
#define CYCLE 5000 // Ciclo de 20ms
#define AV_FULL_IZD 450 // 1.8ms --> Adelante a máxima velocidad
void setup () {
        pinMode (9, OUTPUT); //vinculamos el pin 9 con el servo, lo configuramos como salida
        cli (); //Inhabilita interrupciones
        // Timer 3. Rueda IZDA
        TCCR3A = 0; // Resetea TCCR3A. WGM31 = WGM30 = 0
```

```
        TCCR3B = 0; // Resetea TCCR3B. WGM32 = 0
        TCNT3 = 0; // Timer/Counter Register
        OCR3A = AV_FULL_IZD; // Registro comparador Salida A
        OCR3B = CYCLE; // Registro comparador Salida B
        TCCR3B |= (1<<CS31)|(1<<CS30); // Clk/64 (Preescala 3). CS31 = CS30 = 1
        TIMSK3 |= (1<<OCIE3A)|(1<<OCIE3B); // Timer3 InterruptMask Register. Habilita que la comparación A & B coincida con la interrupción (OCR3A & OCR3B). OCIE3A = OCR3B = 1
        sei (); //Habilita interrupciones
}
ISR (TIMER3_COMPA_vect) { //Rutina de atención a la interrupción. Timer3 A
        digitalWrite (9,LOW);
}
ISR (TIMER3_COMPB_vect) { //Rutina de atención a la interrupción. Timer3 B
        digitalWrite (9,HIGH);
        TCNT3 = 0;
}
void loop() {
        //Especificamos el código que queremos realizar mientras se produce el giro en los motores
}
```

Modelos de motores servo

En este proyecto se han utilizado los dos modelos de motores servo que se indican a continuación, el Futaba S3003 y el SG90 Microservo.

- **Servo Futaba S3003**

Este modelo de servo, que aparece en la figura 2.12, ha sido el elegido para darle movimiento a las ruedas de nuestro vehículo. Los motores han sido trucados y permiten un giro de 360º, lo que nos permite un movimiento continuo para el giro de las ruedas.

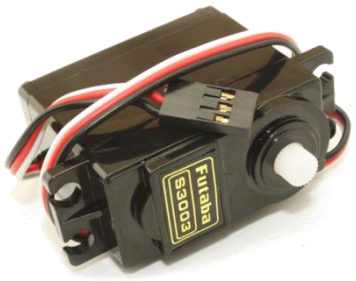

Figura 2.12: Servo Futaba

A pesar de haber sido trucados, mantienen los anchos de pulso originales para su movimiento. Un pulso de 1,5ms hace que el servo esté en una posición de reposo. Los pulsos de anchura de 1,2 y 1,8ms nos dan la máxima velocidad en ambos sentidos. Para que las ruedas del vehículo giren en el mismo sentido, los servos deben girar en sentidos opuestos.

Especificaciones técnicas

- Tensión de trabajo: 4,8-6,0V.
- Control: ancho de pulso para control de 1520µs.
- Pulso requerido: onda cuadrada de 3-5V de amplitud.
- Velocidad de trabajo (4.8V): 0,23s/60 grados.
- Velocidad de trabajo (6V): 0,19s/60 grados.
- Permite ser modificado para girar 360º.
- Tipo de motor: 3 polos de ferrita

- **SG90 Microservo**

Este modelo de servo (véase la figura 2.13) ha sido elegido para darle movimiento al sensor de ultrasonidos dentro de un rango fijo de 180 grados para poder hacer mediciones en zonas concretas.

Figura 2.13: Servo SG90

Para el movimiento de este servo mantenemos los mismos anchos de pulso que se han comentado para el modelo anterior.

Especificaciones técnicas

- Tensión de trabajo: 4,8V (~5V).
- Velocidad de trabajo: 0,1s/60 grados
- Ancho de banda: 10µs.

2.3 Sensores utilizados

Un sensor es un dispositivo capaz de detectar magnitudes físicas o químicas y transformarlas en variables eléctricas. Las variables de instrumentación pueden ser, por ejemplo, la temperatura, la intensidad lumínica, la distancia, la aceleración, la presión, la fuerza, la humedad, etc. Una variable eléctrica puede ser una resistencia eléctrica, una capacidad eléctrica, una tensión eléctrica, una corriente eléctrica, etc.

Características generales de los sensores

- Rango de medida: dominio en la magnitud medida en el que se puede aplicar el sensor.
- Exactitud: error de medida máximo esperado.
- Offset o desviación de cero: valor de la variable salida cuando la variable de entrada es nula.
- Linealidad.
- Sensibilidad: mínima magnitud en la señal de entrada requerida para producir una determinada magnitud en la señal de salida.
- Resolución: mínima variación de la magnitud de entrada que puede detectarse a la salida.

A continuación vamos a desarrollar los sensores empleados en nuestro proyecto, que son: el sensor de ultrasonidos, el sensor de luz (fotodiodo), el sensor de infrarrojos y el sensor de temperatura y humedad.

2.3.1 Sensor de ultrasonidos

Un sensor de ultrasonidos o sensor ultrasónicos son detectores de proximidad que trabajan libres de roces mecánicos y que detectan objetos a distancias que van desde los pocos centímetros hasta los varios metros.

Hemos escogido el modelo HC-SR04 para este proyecto ya que es el más comúnmente utilizado con Arduino (figura 2.14).

Figura 2.14: Sensor de ultrasonidos

Este tipo de sensor emite pulsos ultrasónicos que, al reflejarse en un objeto, produce un eco que es recibido por el sensor y éste lo convierte en señal eléctrica. Estos sensores pueden detectar objetos de diferentes formas, superficies y materiales, y trabajan según el tiempo transcurrido del eco, es decir, la distancia temporal entre el impulso de emisión y el impulso del eco.

En nuestro proyecto utilizamos este sensor para detectar obstáculos en la parte frontal de nuestro vehículo, como se puede ver en la figura 2.15. En el movimiento manual, el sensor se encargará de realizarnos una medida de la distancia hasta el objeto más próximo cuando nosotros le indiquemos; mientras que, en el automático, el vehículo irá desplazándose en línea recta y en el caso de detectar un objeto delante, éste se detendrá para realizar mediciones a ambos lados y determinar qué camino tiene el obstáculo más alejado para proseguir por él.

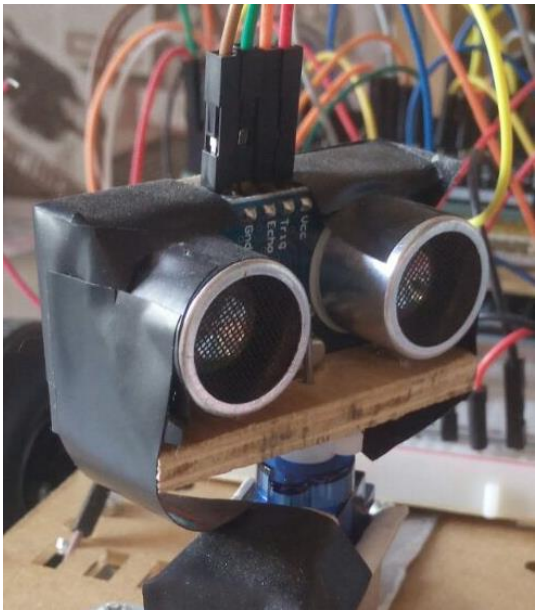

Figura 2.15: Colocación del sensor HC-SR04 en el vehículo.

Dispondremos de otro sensor de proximidad en el “semáforo” (figura 2.16), que nos detectará si el vehículo se encuentra próximo o no a éste y poder encender el led verde y apagar el rojo.

Figura 2.16: Colocación del sensor HC- SR04en el “semáforo”.

Especificaciones técnicas

- Tensión de trabajo: 5V.
- Corriente de trabajo: 15mA.
- Rango de distancia: 2-400cm.
- Precisión: 3mm.
- Frecuencia de trabajo: 40Hz.

Funcionamiento

Se muestra en la figura 2.17. Cuando recibe un pulso en su entrada TRIG, este módulo emite ocho pulsos a 40kHz y recibe los ecos de los mismos. La salida ECHO genera un ancho de pulso proporcional a la distancia al obstáculo.

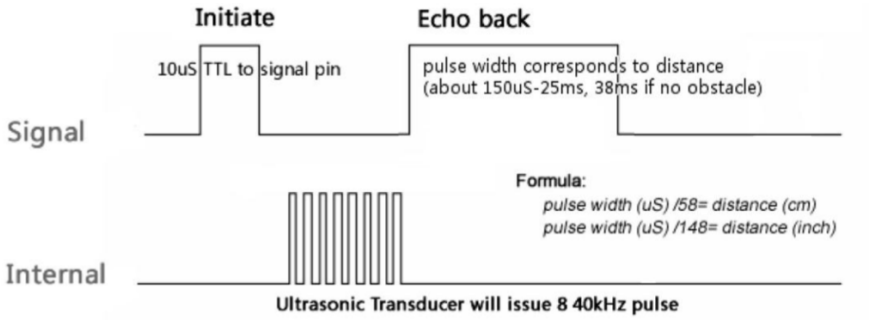

Fig.2.17: Funcionamiento del sensor de ultrasonidos

Para obtener la distancia al obstáculo calcularemos el rango a través del intervalo entre la señal enviada del trigger y la señal recibida del eco. La velocidad del sonido a 20º de temperatura es 340m/s. Como la señal tiene que ir y volver, la distancia será la mitad de la recorrida. La ecuación por la que se obtiene la distancia es la 2.2.

$$distancia\ (cm) = \frac{duración\ (\mu s) * 340\ (m/s)}{2} \qquad \text{Ecuación 2.2}$$

Transformamos la ecuación anterior para obtener los resultados directamente trabajando con microsegundos. Teniendo en cuenta que 340m/s son 0,034cm/µs, la ecuación final es la 2.3.

$$distancia\ (cm) = \frac{duración\ (\mu s)}{29.4\ (\mu s/cm) * 2}$$ Ecuación 2.3

Conexionado

Posee 4 pines de conexión: VCC (5V), TRIG (Trigger pulso de entrada), ECHO (Eco pulso de salida), y GND. Los cables rojo y negro son la alimentación (5V y GND), y los cables azul y amarillo corresponden con TRIG y ECHO. Los pines para ECHO y TRIG son los que se hubieran configurado en el Arduino como entrada y salida respectivamente. Se puede observar la conexión en la figura 2.18.

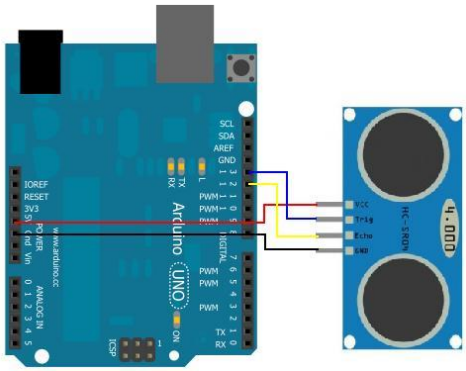

Figura 2.18: Conexionado del sensor de ultrasonidos

Código

El código de Arduino para este sensor es el siguiente:

```
long duration, cm; //creamos variables para todo el código en las que almacenar los valores
void setup(){
        Serial.begin (9600); //Abrimos comunicación con el puerto serie para ver por pantalla las medidas. La consola se abre a velocidad de 9600 baudios
        pinMode (13, OUTPUT); //Configuramos el pin 13 como salida para TRIG
        pinMode (12, INPUT); //Configuramos el pin 12 como entrada para ECHO
}
void loop(){
        digitalWrite (13, LOW);  //se activa y desactiva el TRIG para emitir los pulsos
        delayMicroseconds (2);
        digitalWrite (13, HIGH);
        delayMicroseconds (5);
        digitalWrite (13, LOW);
        duration = pulseIn (12, HIGH); //obtenemos el pulso de entrada
        cm = microsecondsToCentimeters (duration); //calculamos los cm de distancia con la función microsecondsToCentimeters
        Serial.print ("Milisegundos: "); //mostramos por el terminal milisegundos
        Serial.print (duration); //mostramos por el terminal el tiempo transcurrido
        Serial.print ("  Distancia estimada: "); //mostramos por el terminal distancia estimada
        Serial.print (cm); //mostramos por el terminal la distancia recorrida
        Serial.println (" cm"); //mostramos por el terminal cm
        delay(1000);
}
long microsecondsToCentimeters (long microseconds){
```

```
        // La velocidad del sonido a 20º de temperatura es 340 m/s o 29 microsegundos por centímetro.
        // La señal tiene que ir y volver, por lo que la distancia a la que se encuentra el objeto es la mitad de la recorrida.
        return (microseconds / 29 /2) ; //devuelve el cálculo de cm en función del tiempo que se le pasa a la función
}
```

En la figura 2.19 vemos un ejemplo de funcionamiento en el cual separamos el sensor poco a poco de una pared y va realizando medidas de distancia respecto de ésta.

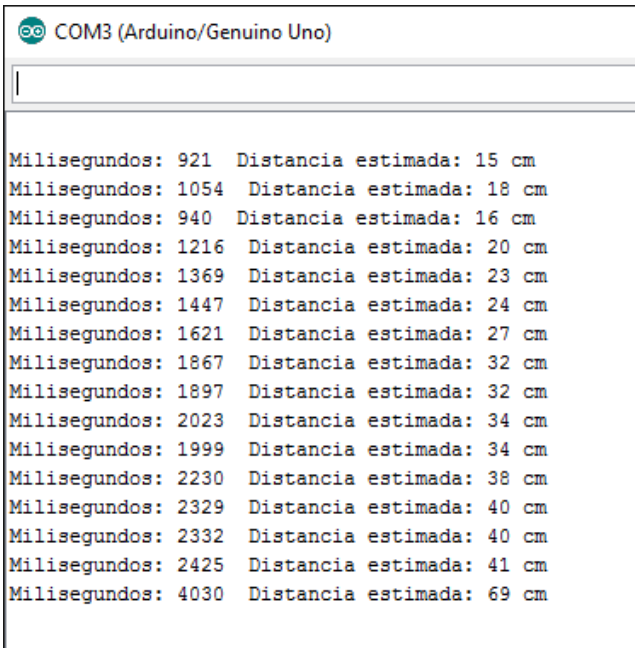

Figura 2.19: Medidas del sensor de ultrasonidos

2.3.2 Conversor de intensidad de luz a frecuencia

El TCS230 (figura 2.20) es un sensor programable fotoeléctrico de tipo fotodiodos que convierte la intensidad de luz en frecuencia.

Figura 2.20: Sensor TCS230

Un fotodiodo es un semiconductor construido con una unión PN, sensible a la incidencia de la luz visible o infrarroja. Se polariza inversamente para producir una cierta circulación de corriente cuando es excitado por la luz.

En nuestro proyecto utilizaremos dos sensores, uno para detectar los colores que habrá en el suelo y otro para detectar cuándo pasa el "semáforo" de rojo a verde para que el vehículo siga desplazándose. En la figura 2.21 podemos ver el sensor TCS230 en la parte frontal del vehículo, que se utilizará para detectar el color del "semáforo". Más adelante indicaremos dónde se sitúa el otro sensor que detectará los colores en el suelo.

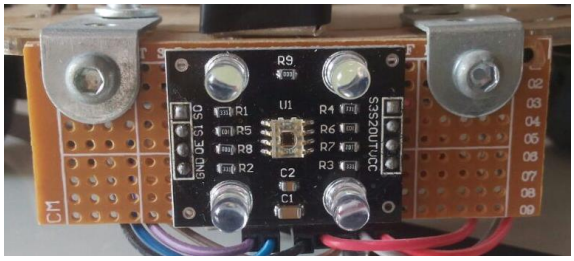

Figura 2.21: Posición del sensor TCS230 que detecta el "semáforo" en el vehículo.

El sensor TCS230 combina fotodiodos configurables de silicio y un conversor de corriente a frecuencia en un circuito integrado monolítico de CMOS. La señal de salida es una onda cuadrada (50% de ciclo de trabajo) con frecuencia directamente proporcional a la intensidad de luz (irradiación). La escala total de frecuencia de salida puede preconfigurarse con uno de los tres valores disponibles utilizando dos pines de entrada de control. Las entradas y salidas digitales permiten interactuar directamente con un microcontrolador. La habilitación de la salida (OE) sitúa la salida en estado de alta impedancia.

El conversor lee una matriz de 8 x 8 fotodiodos. 16 fotodiodos tienen filtros azules, 16 los tienen verdes, otros 16 los tienen rojos y otros 16 los tienen transparentes o no tienen filtros. Los cuatro tipos (colores) de fotodiodos son intercalados para minimizar el efecto de la no uniformidad de la irradiación incidente; todos los fotodiodos del mismo color son conectados en paralelo, y en función de la selección de los pines se usan unos u otros.

Especificaciones técnicas

- Tensión de trabajo: 2,7-5,5V.
- Escala de frecuencias máximas de trabajo en el dispositivo sin saturación:
  - S0 = H, S1 = H: 500-600kHz.
  - S0 = H, S1 = L: 100-120kHz.
  - S0 = L, S1 = H: 10-12kHz.
- La respuesta frecuencial la podemos comprobar en la figura 2.22, en la que podemos observar que la banda infrarroja nos va a interferir en las medidas.

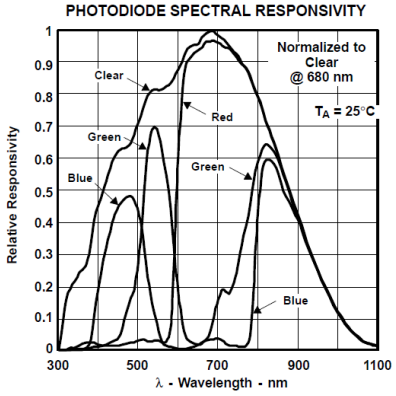

Figura 2.22: Respuesta frecuencial del TCS230

En la tabla 2.1 vemos los pines del módulo TCS230 y en la 2.2 la configuración de éstos.

TERMINAL NAME	NO.	E/S	DESCRIPCIÓN
GND	**4**		Alimentación suministrada a tierra. Todas las tensiones se referencian a GND.
OE	**3**	E	Habilita f_O (activo a nivel bajo).
OUT	**6**	S	Frecuencia de salida (f_O).
S0, S1	**1, 2**	E	Pines de selección para escalar la frecuencia de salida.
S2, S3	**7, 8**	E	Pines de selección del tipo de fotodiodo.
V_{DD}	**5**		Tensión de alimentación.

Tabla 2.1: Pines del TCS230

S0	S1	ESCALA FRECUENCIA DE SALIDA (f_O)	S2	S3	TIPO DE FOTODIODO
L	**L**	Apagado	**L**	**L**	Rojo
L	**H**	2%	**L**	**H**	Azul
H	**L**	20%	**H**	**L**	Transparente (sin filtro)
H	**H**	100%	**H**	**H**	Verde

Tabla 2.2: Configuración pines del TCS230

Funcionamiento

El sensor funciona basándose en las frecuencias medidas según el tipo de filtro que se haya configurado. Las frecuencias están directamente relacionadas con la intensidad de la luz, y por ello se utiliza una medición por separado de los colores primarios rojo, verde y azul. Esta composición se llama RGB (Red Green Blue, Rojo Verde Azul) y la combinación de las diferentes intensidades de luz de cada color da como resultado un color. Cada color varía del 0 al 255; cuando los tres valen 0, el color correspondiente es el negro (ausencia de color), y si los tres valen su máximo de 255, se representa el blanco (mezcla de todos los colores).

En la figura 2.23 podemos ver el diagrama funcional de bloques del sensor. En ella se observa que la luz es captada por la matriz de diodos que se configura con los pines S2 y S3, como se ha comentado anteriormente. La señal captada se transmite al conversor de corriente a frecuencia, donde la escala de frecuencia de salida se configura con los pines S0 y S1, como también se ha visto anteriormente. Este conversor divide la señal por los valores de escala y genera un tren de pulsos de ancho ajustable según la escala escogida, haciendo que las salidas sean señales cuadradas del 50% del ciclo de trabajo. Todo esto es debido a que la división de la frecuencia de salida está compuesta por la cuenta de pulsos de la frecuencia principal interna, haciendo que el período final de salida represente una media de múltiples períodos de la frecuencia principal.

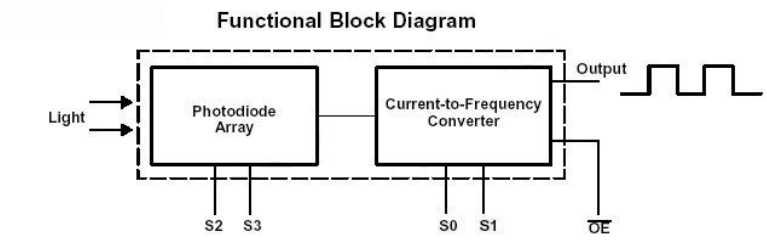

Figura 2.23: Diagrama de bloques del TCS230

Conexiones

Alimentamos este módulo uniendo sus pines 5V y GND a los pines correspondientes del Arduino, y el resto de conexiones las realizamos en función de cómo hayamos programado los pines del Arduino, tal y como se indica en la figura 2.24.

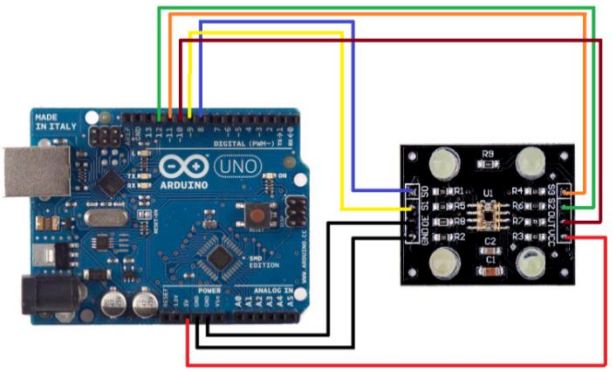

Figura 2.24: Conexionado del sensor TCS230

Código

Para el uso de este sensor necesitamos incluir una librería adicional para el manejo de *Timers*: "*TimerOne*".

```
#include <TimerOne.h>
#define S0    6
#define S1    5
#define S2    4
#define S3    3
#define OUT   2
int g_count = 0;   // cuenta la frecuencia
int g_array[3];    // almacena el valor RGB
int g_flag = 0;    // filtro de la cola de RGB (filter of RGB queue)
float g_SF[3];       // guarda el factor de escala RGB
// Inicia TCS230 y la configuración de frecuencia
void TSC_Init () {
        pinMode (S0, OUTPUT); //pin configurado de salida para activar la escala de frecuencia
        pinMode (S1, OUTPUT); //pin configurado de salida para activar la escala de frecuencia
        pinMode (S2, OUTPUT); //pin configurado de salida para activar los filtros de colores
        pinMode (S3, OUTPUT); //pin configurado de salida para activar los filtros de colores
        pinMode (OUT, INPUT); //pin configurado de salida para la salida del sensor
        digitalWrite (S0, LOW);  //escala de frecuencia de salida al 2%
        digitalWrite (S1, HIGH);
}
void TSC_Count () { //Contador de frecuencia
        g_count ++ ;
}
void TSC_Callback () {
        switch (g_flag) {
                case 0:
                        Serial.println ("->WB Start");
                        TSC_WB (LOW, LOW); // Filtro configurado para rojo
                        break;
                case 1:
                        Serial.print ("->Frequency R=");
                        Serial.println (g_count); //Muestra la frecuencia medida del color
                        g_array[0] = g_count;
```

```
                    TSC_WB (HIGH, HIGH); // Filtro configurado para verde
                    break;
            case 2:
                    Serial.print ("->Frequency G=");
                    Serial.println (g_count); //Muestra la frecuencia medida del color
                    g_array[1] = g_count;
                    TSC_WB (LOW, HIGH); // Filtro configurado para azul
                    break;
            case 3:
                    Serial.print ("->Frequency B=");
                    Serial.println (g_count); //Muestra la frecuencia medida del color
                    Serial.println ("->WB End");
                    g_array[2] = g_count;
                    TSC_WB (HIGH, LOW); // Filtro configurado para transparente o sin filtro
                    break;
            default:
                    g_count = 0;
                    break;
        }
}
void TSC_WB (int Level0, int Level1) { //Balance de blancos (White Balance)
        g_count = 0;
        g_flag ++;
        TSC_FilterColor (Level0, Level1); //Llama a la función de selección de filtro
        Timer1.setPeriod (1000000); // Configura un período de 1s
}
// Selecciona el color del filtro
void TSC_FilterColor (int Level01, int Level02) {
        if (Level01 != 0)
                Level01 = HIGH;
        if (Level02 != 0)
                Level02 = HIGH;
        digitalWrite (S2, Level01);
        digitalWrite (S3, Level02);
}
void setup() {
        TSC_Init (); //Llama a la función de inicializar el TCS230
        Serial.begin (9600); //Comunicación serie con el monitor
        Timer1.initialize (); //La interrupción se producirá cada segundo
        Timer1.attachInterrupt (TSC_Callback); //La interrupción llamará a la función TSC_Callback
        attachInterrupt (0, TSC_Count, RISING); //El primer parámetro representa el número de interrupción, el segundo la función que llama cuando se produce la interrupción, y el tercero nos dice que la condición de disparo es en el flanco de subida (cuando pasa de LOW a HIGH)
        delay (4000);
        for (int i=0; i<3; i++)
                Serial.println(g_array[i]); //Muestra las frecuencias recogidas para la configuración
        g_SF[0] = 255.0/ g_array[0]; //Factor de escala R
        g_SF[1] = 255.0/ g_array[1]; //Factor de escala G
        g_SF[2] = 255.0/ g_array[2]; //Factor de escala B
        Serial.println (g_SF[0]); //Muestra los factores de escala RGB
        Serial.println (g_SF[1]);
        Serial.println (g_SF[2]);
}
void loop() {
        g_flag = 0;
        for (int i=0; i<3; i++)
                Serial.println (int (g_array[i] * g_SF[i])); //Muestra los valores medidos por el factor obtenido al principio para tener el resultado del RGB
        delay (4000);
}
```

Los resultados del código anterior los vamos a comentar paso a paso. En primer lugar, en la figura 2.25, vemos que el sensor se configura sobre un fondo blanco, para después medir diferentes colores, figuras 2.26 y 2.27.

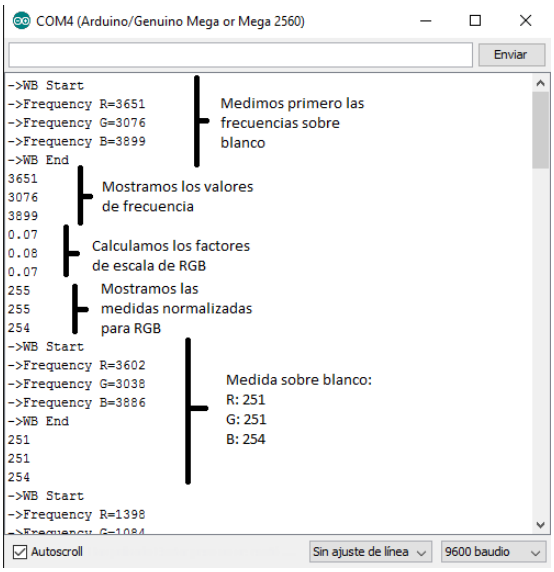

Figura 2.25: Configuración del TCS230

A continuación, en la figura 2.26, vemos los resultados de las mediciones para diferentes colores.

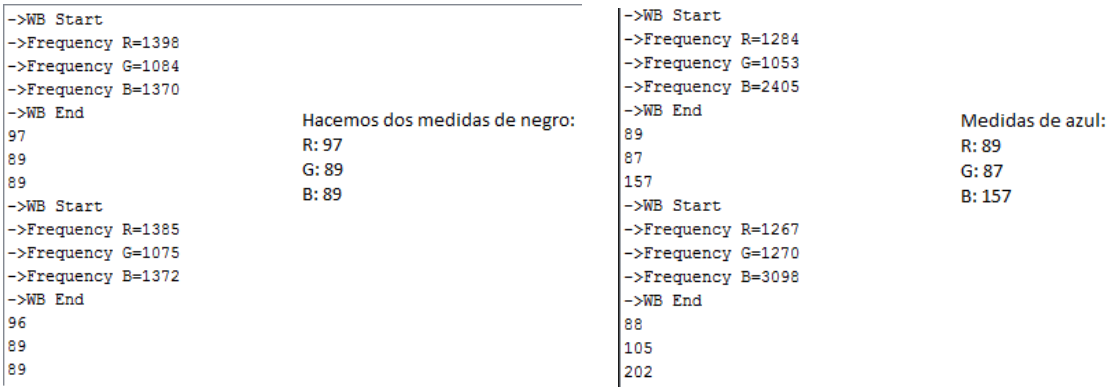

Figura 2.26: Medidas de los colores negro y azul

Por último, en la figura 2.27, vemos una medición del color rojo.

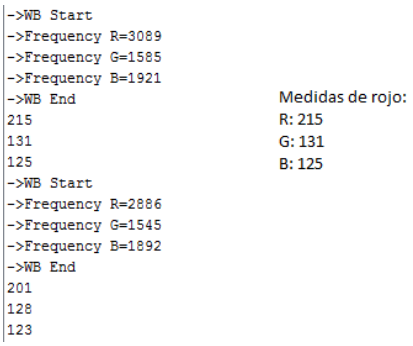

Figura 2.27: Medida del color rojo

En las figuras anteriores hemos podido observar los resultados de la programación del sensor. No son muy exactos, ya que la banda de infrarrojos interfiere, como pudimos observar en la figura 2.22, a lo que habría que añadir el que la luz natural introduce interferencias en las medidas, pero podemos dar por satisfactorias estas mediciones.

2.3.3 Sensor de infrarrojos

Un sensor infrarrojo es un dispositivo optoeléctrico capaz de medir la radiación electromagnética infrarroja de los cuerpos en su campo de visión. Existen dos tipos de sensores: los pasivos y los activos. Los sensores pasivos únicamente están formados por un fototransistor que mide las radiaciones provenientes de los objetos, mientras que un sensor activo combina un emisor (diodo infrarrojo) y un receptor (fototransistor) próximos entre ellos, y que normalmente forman parte del mismo circuito integrado.

Por todo ello, se ha escogido el modelo TCRT5000 (figura 2.28), un sensor infrarrojo que incluye un diodo emisor de infrarrojos y un fototransistor en un encapsulado que bloquea la luz visible. Su papel en nuestro proyecto será el de detectar el cambio de luminosidad generado por el paso de una superficie blanca a una negra o viceversa, lo que hará que el vehículo rectifique la trayectoria evitando "pisar" las líneas negras. Esto permitirá que el vehículo siga el camino trazado por ellas, ya que dispondrá de dos sensores para medir a ambos lados y mantener así un movimiento lineal.

Figura 2.28: Sensor TRCT5000

Especificaciones técnicas

- Tensión máxima de trabajo del diodo emisor: 1,5V.
- Corriente máxima de trabajo del diodo emisor: 60mA.
- Tensión de colector: 5V.
- Longitud de onda emisor: 950nm.
- Filtro bloqueador de luz de día.
- Rango de distancias de trabajo: 0,2-15mm.

Para el uso correcto de este sensor hay que hacer un acondicionador de señal para obtener un valor de tensión en función de la corriente obtenida por el reflejo del infrarrojo.

Acondicionamiento de los sensores

Tanto el diodo emisor como el fototransistor receptor necesitan un circuito que les limite las corrientes para que éstos no sufran daños.

- Acondicionador del diodo emisor

El circuito a diseñar se compone en este caso de un diodo emisor y una resistencia en serie (se puede ver en la figura 2.29).

Figura 2.29: Circuito acondicionador diodo.

Como máximo puede pasar una corriente de 60mA por el diodo emisor, así que si se alimenta a 5V, la resistencia a colocar tendrá una caída de tensión de 3,5V, ya que la caída de tensión en el diodo es de 1,5V. La corriente máxima nunca es alcanzada, puesto que la corriente que proporciona el pin del Arduino limita en 50mA. Con la Ley de Ohm calculamos el valor de la resistencia con valores de corriente menores a los 50mA que proporciona Arduino para evitar posibles sobrecargas en el diodo, ecuaciones 2.4 y 2.5.

$$\text{I} = 40\text{mA} \; ; \; V = R * I \; ; \; R = \frac{V}{I} = \frac{3{,}5}{0{,}04} = 87{,}5\Omega \qquad \text{Ecuación 2.4}$$

$$\text{I} = 30\text{mA} \; ; \; V = R * I \; ; \; R = \frac{V}{I} = \frac{3{,}5}{0{,}03} = 116{,}66\Omega \qquad \text{Ecuación 2.5}$$

Decidimos utilizar una resistencia de 100Ω ya que es un valor comprendido entre los calculados en las ecuaciones 2.4 y 2.5.

- Acondicionador del fototransistor receptor

El circuito acondicionador a implementar es parecido al anterior y se puede ver en la figura 2.30. Únicamente consta de una resistencia en el colector del fototransistor, el cual está configurado en modo emisor común. La resistencia de colector será elegida de manera experimental y para ello vamos a medir de forma analógica nuestro valor de tensión, que estará comprendido entre 0V (superficie blanca) y 5V (superficie negra).

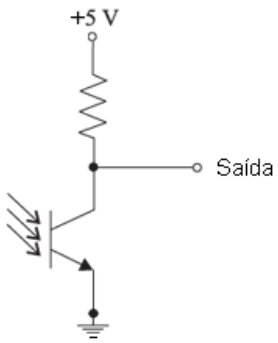

Figura 2.30: Circuito acondicionador fototransistor.

A mayor resistencia de colector tendremos una salida menor, por lo que comenzamos a realizar medidas con una resistencia de 47kΩ. Nos ayudamos de potenciómetros de 10kΩ para observar los resultados al bajar la resistencia. Las medidas las hemos realizado con el osciloscopio del laboratorio de la facultad, obteniendo los siguientes resultados:

En la figura 2.31 observamos que la salida obtenida para 47kΩ es prácticamente nula.

Figura 2.31: medida con R=47kΩ

En la figura 2.32 podemos ver cómo mejora la salida al disminuir las impedancias a 27kΩ (izquierda) y 12 kΩ (derecha).

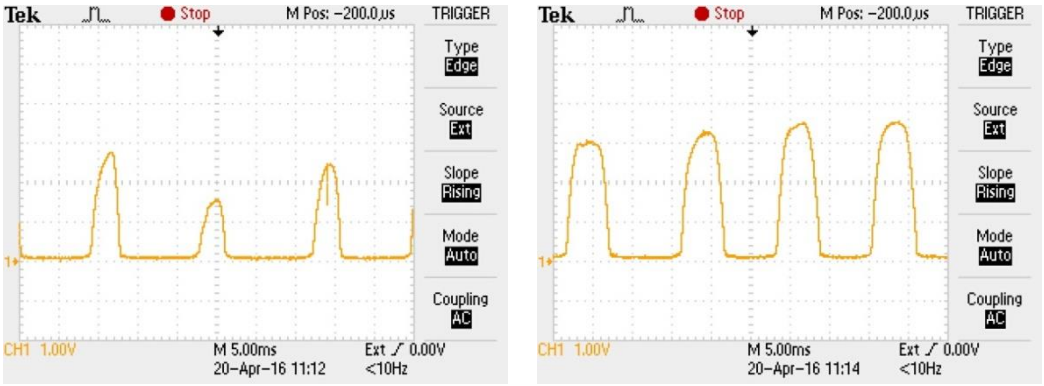

Figura 2.32: medidas con R=27kΩ y con R=12kΩ

Finalmente, en la figura 2.33 podemos ver la mejora al utilizar una impedancia de 6,8kΩ.

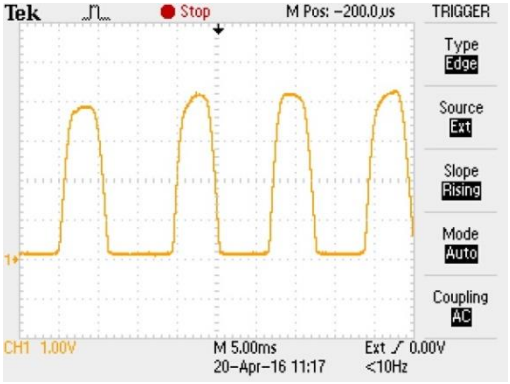

Figura 2.33: medida con R=6,8kΩ.

Como se observa en esta última figura, obtenemos una salida algo superior a 4V, por lo que decidimos realizar el circuito acondicionador con esta resistencia ya que obteníamos una señal de salida estable y bastante exacta.

- Diseño PCB

Puesto que una de las funciones del proyecto es que el vehículo evite líneas y detecte colores en el suelo, se ha decidido crear una PCB (Printed Circuit Board, Placa de Circuito Impreso) para tener los sensores TCRT5000 y el TCS230 juntos y más próximos al suelo.

Para ello hemos utilizado unas placas para circuitos con cobre por una de las caras para soldadura. En ella hemos soldado los diferentes sensores y los componentes para sus acondicionadores de señal, y también los pines donde conectar posteriormente los cables para alimentar la placa y comunicarla con el Arduino (figura 2.34).

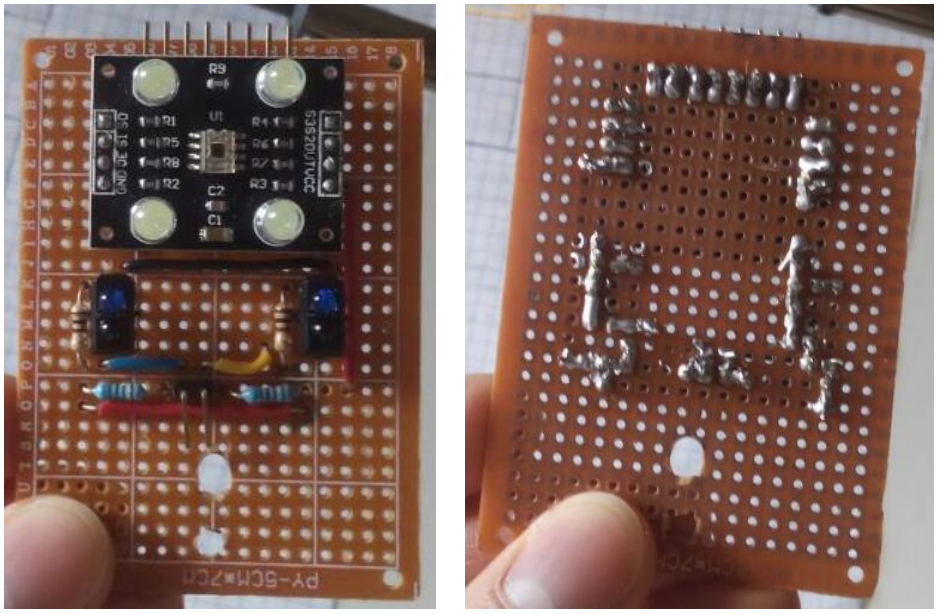

Figura 2.34: PCB con los sensores y componentes.

En la figura 2.35 podemos ver que la PCB se sitúa en la parte inferior del vehículo.

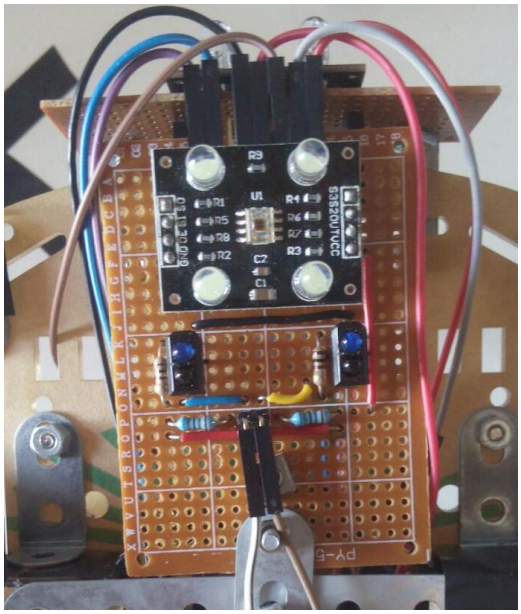

Figura 2.35: Colocación de la PCB en el vehículo.

Funcionamiento

Las salidas que obtenemos con los acondicionadores se encuentran conectadas a los pines analógicos del Arduino, por lo que, si la reflexión es pequeña, el fototransistor estará cortado y la medición rondará 0 (0V, zona blanca), mientras que si la luz reflejada es elevada, el fototransistor conducirá y el valor medido rondará 1024 (5V, zona negra).

Conexionado

Los pines de alimentación irán conectados al pin de 5V, con sus respectivas resistencias, y al pin GND del Arduino. Las medidas las realizaremos con los pines analógicos, ya que nos devolverá valores comprendidos entre 0 y 5V, por lo que la salida irá conectada a uno de estos pines. La conexión se puede ver en la figura 2.36, donde podemos comprobar que las alimentaciones van al ánodo del diodo y al colector del fototransistor. Las resistencias de la figura 2.36 no corresponden con las resistencias utilizadas en la realidad y calculadas anteriormente.

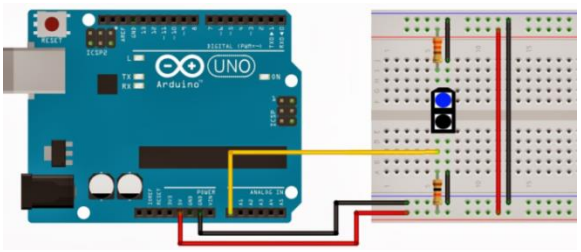

Figura 2.36: Conexionado del sensor TCRT5000

Código

El código de este sensor se basa en leer la entrada analógica.

```
int linea; //Creamos una variable para almacenar las medidas
void setup() {
        Serial.begin(9600); //Abrimos la comunicación con el monitor
}
void loop(){
        linea = analogRead (A0); //Leemos de forma analógica la medida
        Serial.println ("Medida:"); //Mostramos por el monitor la palabra Medida
        Serial.println (linea); //Mostramos por el monitor el valor medido
        delay (500);
}
```

En la figura 2.37 podemos ver perfectamente el cambio que se produce de medir sobre una superficie blanca a una negra, explicado en el apartado de funcionamiento.

```
Medida:
18
Medida:
17
Medida:
18
Medida:
19
Medida:
1023
Medida:
1023
```

```
Medida:
1023
Medida:
1023
Medida:
1023
Medida:
17
Medida:
18
Medida:
17
```

Figura 2.37: Medidas del sensor TCRT5000

2.3.4 Sensor de temperatura y humedad: DHT11

El DHT11 (figura 2.38) es un sensor de temperatura y humedad que pertenece a la familia de sensores DHT. Estos sensores están compuestos por dos partes, un sensor capacitivo de humedad y un termistor. También posee un chip básico interno que realiza una conversión analógico-digital, devolviendo la medida digital con la temperatura y la humedad.

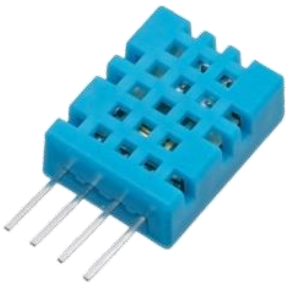

Figura 2.38: Sensor DHT11

Un termistor es un sensor resistivo de temperatura y su funcionamiento se basa en la variación de la resistencia que presenta un semiconductor con la temperatura. Esta variación no es lineal.

Dentro de la familia de DHT existen dos modelos: DHT11 y DHT22. En nuestro proyecto utilizaremos el modelo DHT11, ya que es el más sencillo de utilizar, y tendrá la función de medirnos la temperatura y humedad del ambiente, en ambos modos de funcionamiento, para después transmitirlas a la aplicación móvil y mostrarlas por pantalla.

En nuestro vehículo este sensor irá situado sobre una *Protoboard* en la parte superior de este como se puede ver en la figura 2.39.

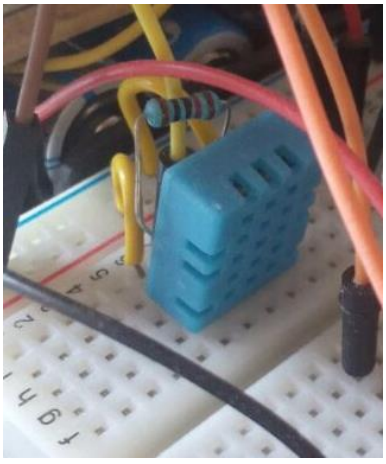

Figura 2.39: Posición del sensor DHT11 en el vehículo.

Especificaciones técnicas

- Tensión de trabajo: 3-5V.
- Rango de temperatura: 0º-50º con 5% de precisión.
- Rango de humedad: del 20% al 80% con 5% de precisión.
- 1 muestra por segundo.
- Devuelve la medida en ºC.

Funcionamiento

El sensor DHT11 utiliza su propio protocolo de comunicación a través de un solo hilo. Al ser simple el protocolo, permite ser utilizado con los pines de un microcontrolador. El microcontrolador inicia la comunicación con el DHT11 manteniendo la línea de datos en estado bajo durante al menos 18 ms. Luego, el DHT11 envía una respuesta con un pulso a nivel bajo de 80 µs y después deja "flotar" la línea de datos por otros 80 µs. Los datos binarios se codifican según la longitud del pulso alto. Todos los bits comienzan con un pulso bajo de 50 µs. En nuestra librería aprovechamos el pulso bajo en cada bit para sincronizar con la señal del DHT11. Después viene un pulso alto que varía según el estado lógico o el valor del bit que el DHT11 desea transmitir. Se utilizan pulsos de 26-28 microsegundos para un "0" y de 70 microsegundos para un "1". Los pulsos se repiten hasta un total de 40 bits. En la figura 2.40 podemos ver este funcionamiento.

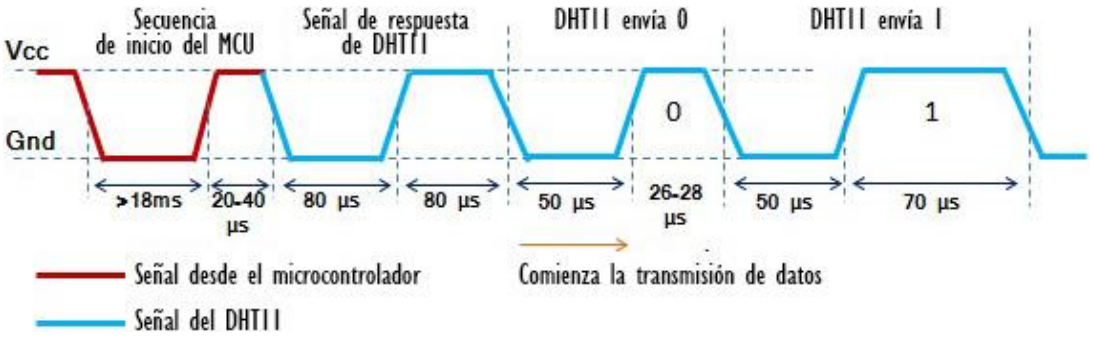

Figura 2.40: Funcionamiento del sensor DHT11

En cuanto a los datos que se transmiten, su interpretación es como sigue:

- Se transmiten 40 bits (5 bytes) en total.
- El primer byte que recibimos es la parte entera de la humedad relativa (RH).
- El segundo byte es la parte decimal de la humedad relativa (no se utiliza en el DHT11, siempre es 0).
- El tercer byte es la parte entera de la temperatura.
- El cuarto byte es la parte decimal de la temperatura (no se utiliza en el DHT11, siempre es 0).
- El último byte es la suma de comprobación (checksum), resultante de sumar todos los bytes anteriores.

Conexionado

El sensor dispone de 4 pines: VDD, datos, NC, GND. Como se indica en la figura 2.41, los cables rojo y negro son la alimentación (5V y GND), y el cable azul corresponde con la patilla de medición. Este pin será configurado como salida. Cuando el cable de conexión de datos es menor de 20 metros, se recomienda una resistencia de 5kΩ.

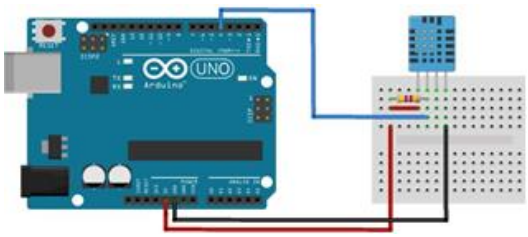

Figura 2.41: Conexionado del DHT11

Código

Incluimos la librería del propio sensor, que facilita las medidas en nuestro código.

```
#include <DHT11.h> //incluimos librería del sensor
DHT11 dht11 (4);  //creamos la conexión del sensor en el pin 4
void setup () {
        Serial.begin (9600);
}
void loop (){
        int err;
        float temp, hum; //variables donde almacenar los datos
        if ((err=dht11.read(hum, temp))==0) { //cuando no haya error, se realizará la medida
                Serial.print ("Temperatura:");
                Serial.print (temp);
                Serial.print (" humedad:");
                Serial.print (hum);
                Serial.println ();
        }
        else { //en caso de algún error, nos dirá el número del error
                Serial.println ();
                Serial.print ("Error No :");
                Serial.print (err);
                Serial.println ();
        }
        delay (DHT11_RETRY_DELAY); //retardo para leer
}
```

En la figura 2.42 podemos ver el resultado de la medición.

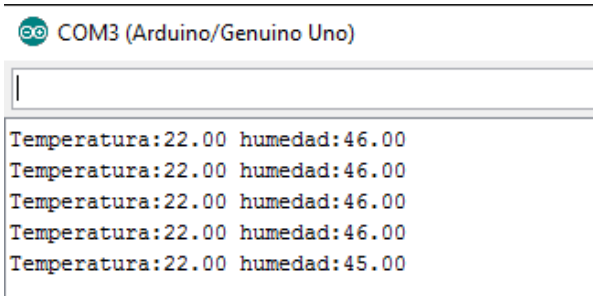

Figura 2.42: Resultados de la medición del DHT11

2.4 Comunicación: módulo Bluetooth HC-05

Una vez comentados los aspectos relativos a las placas de Arduino utilizadas en el vehículo, los motores y los sensores, veremos a continuación cómo conectaremos el vehículo con dispositivos móviles a través de un módulo Bluetooth, en concreto el HC-05, pero antes de hablar de este módulo sería preceptivo comentar algo sobre el Bluetooth.

Se denomina Bluetooth al protocolo de comunicaciones diseñado especialmente para dispositivos de bajo consumo, que requieren corto alcance de emisión y basados en transceptores de bajo coste.

Los dispositivos que incorporan este protocolo pueden comunicarse entre sí cuando se encuentran dentro de su alcance. Las comunicaciones se realizan por radiofrecuencia, de forma que los dispositivos no tienen por qué estar alineados y pueden incluso estar en habitaciones

separadas si la potencia de transmisión es suficiente. Estos dispositivos se clasifican comúnmente según su capacidad de canal, tal como se observa en la tabla 2.3:

Versión	Ancho de banda
Versión 1.2	1 Mbit/s
Versión 2.0 + EDR	3 Mbit/s
Versión 3.0 + HS	24 Mbit/s
Versión 4.0	32 Mbit/s

Tabla 2.3: Versiones de Bluetooth según su canal

Para utilizar Bluetooth, un dispositivo debe implementar alguno de los perfiles Bluetooth. Éstos definen el uso del canal Bluetooth y cómo canalizar hacia el dispositivo que se quiere vincular.

La conexión entre el vehículo y la App la realizaremos con el módulo HC-05 (figura 2.43). Este módulo tiene la capacidad de ser maestro o esclavo. La configuración como maestro significa que realiza él la conexión, mientras que siendo esclavo espera a que se conecten con él.

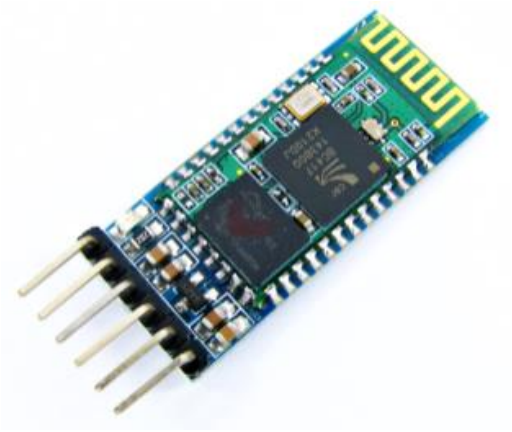

Figura 2.43: Modulo CH-05

Dispone de 6 pines: dos para alimentación (GND y VCC), dos para la transmisión y recepción de datos (TXD y RXD), uno llamado STATE (no utilizado) y otro llamado KEY (para entrar en el modo configuración).

El módulo dispone de tres modos de trabajo que son indicados con el parpadeo de un Led:

- Si el led parpadea constantemente sin parar indica que está esperando una conexión.
- Si está dos segundos encendido y otros dos segundos apagado, se encuentra en modo de comandos AT.
- Si parpadea dos veces y se mantiene apagado tres segundos y vuelve a parpadear dos veces indica que está conectado a algún dispositivo.

La configuración del módulo se realiza mediante los comandos AT. Estos comandos son un lenguaje desarrollado por la compañía Hayes Communications que prácticamente se convirtió en estándar abierto de comandos para configurar y parametrizar módems. Los caracteres <AT>, que preceden a todos los comandos significan <Atención>, lo que hizo que se conocieran a estos comandos como comandos AT.

El hecho de que podamos manejar los movimientos del vehículo a través de la aplicación del móvil mediante Bluetooth amplía enormemente las posibilidades de actuación del vehículo, proporcionando más independencia al usuario, mayor comodidad, etc., y al mismo tiempo obtener información del vehículo en tiempo real de las mediciones de los sensores, posición del vehículo, velocidad de las ruedas, etc.

En el vehículo se colocará en una Protoboard situada sobre éste, como se puede ver en la figura 2.44.

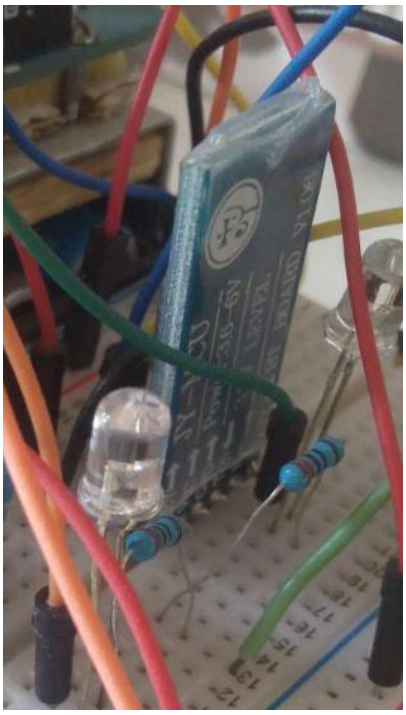

Figura 2.44: Colocación del módulo Bluetooth sobre el vehículo.

Especificaciones técnicas

- Tensión de trabajo: 5V.
- Versión Bluetooth: 2.0.
- Sensibilidad: -80dBm.
- Interfaz UART con velocidad programable.
- Frecuencia de trabajo: 2,4GHz.
- Velocidades soportadas (baudios): 9600, 19200, 38400, 57600, 115200, 230400, 460800.
- Antena integrada.

Funcionamiento

En primer lugar se debe configurar el módulo Bluetooth mediante los comandos AT. Una vez realizada la configuración de velocidad, nombre de dispositivo, etc., procedemos a realizar transmisiones de datos para comprobar que su funcionamiento sea correcto.

Conexiones

Para entrar en el modo AT y poder configurarlo, debemos conectar el módulo con el Arduino (desconectado del PC, sin alimentación) a través de los pines:

- HC-05 GND — Arduino GND Pin
- HC-05 VCC (5V) — Arduino 5V
- HC-05 TX — Arduino Pin 10 (soft RX)
- HC-05 RX — Arduino Pin11 (soft TX)
- HC-05 Key (PIN 34) — Arduino Pin 9

Las conexiones de los pines 10 y 11 del Arduino no importa a qué pines del módulo se realicen, lo importante es que el pin RX del módulo HC-05 se conecte a un pin configurado en Arduino como TX, y el pin TX del módulo vaya a otro pin configurado como RX en el Arduino. Una vez conectado todo, quitamos el pin VCC del módulo y procedemos a conectar el Arduino al PC y cargarle el programa. Tras ello, conectamos de nuevo VCC y tenemos que comprobar lo comentado anteriormente: que el led parpadee dos segundos y se apague otros dos. Debemos cargarlo así para que el pin KEY esté en HIGH, lo que hace que se inicie directamente el modo configuración. Abrimos el Monitor Serial del entorno de desarrollo de Arduino y seleccionamos la opción "Ambos NL & CR", que agrega un retorno de carro y un espacio en blanco después de lo que se envía.

Código

En primer lugar veremos el código de la configuración y los resultados, y posteriormente el código de la comunicación y sus resultados. En ambos códigos vamos a utilizar una librería adicional para crear un "objeto" para el Bluetooth, el cual configuraremos para su uso.

- **Configuración**

```
#include <SoftwareSerial.h>
SoftwareSerial BTSerial(10, 11); // RX | TX
void setup() {
        pinMode (9, OUTPUT); //Este pin se pone a nivel bajo como salida para entrar en modo AT para realizar la configuración
        digitalWrite (9, HIGH);
        Serial.begin (9600);
        Serial.println ("Enter AT commands:");
        BTSerial.begin (38400); //Velocidad por defecto del módulo de Bluetooth
}
void loop() {
//Lee del Serial3, donde está el módulo, lo que envía el módulo al Arduino
        if (BTSerial.available())
                Serial.write(BTSerial.read());
// Lee del Serial, donde está el Arduino, lo que envía el Arduino al módulo
        if (Serial.available())
                BTSerial.write(Serial.read());
}
```

Comenzamos a configurarlo mediante el uso de los comandos AT. Primero enviamos un mensaje escribiendo "AT" para comprobar si todo es correcto y nos responde que sí. A partir de

aquí le preguntamos el nombre (“AT+NAME”), la versión (“AT+VERSION”), su dirección (“AT+ADDR”), modo de trabajo, 0 esclavo y 1 maestro (“AT+ROLE”), su contraseña (“AT+PSWD”) y sus parámetros de serie, ratio de baudios, bit de stop y bit de paridad (“AT+UART”). Las respuestas las podemos ver en la figura 2.45.

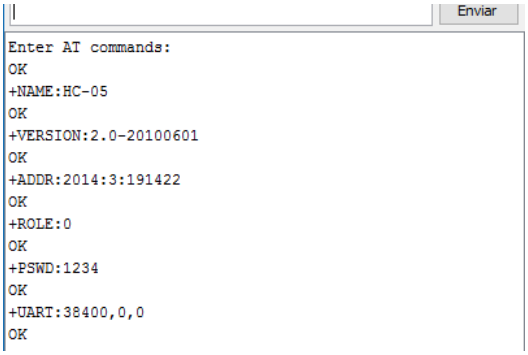

Figura 2.45: Comandos AT y respuestas obtenidas

- **Comunicación**

```
#include <SoftwareSerial.h>
SoftwareSerial BTSerial(10, 11); // RX | TX
byte byteread;
void setup() { // Open serial communications and wait for port to open:
        Serial.begin(9600);
        BTSerial.begin(9600);
        byteread = 0;
}
void loop() {
        while (BTSerial.available()){
                byteread = BTSerial.read();
                Serial.write(byteread); //lo recibe en codigo ASCII y lo muestra decodificado
        }
}
```

Una vez realizada la configuración de nuestro dispositivo, hacemos una conexión para comprobar que nos llegan los datos que enviamos desde el dispositivo conectado. Para realizar pruebas, nos descargamos una aplicación que hace las veces de terminal Bluetooth y nos permite enviar lo que nos interese; esta aplicación es “Bluetooth Serial Controller”. Al abrir los puertos de comunicación, debemos indicar que la comunicación del Bluetooth vaya a tasa de 38400 bits/s, dado que es lo que nos respondió al preguntarle con los Comandos AT. Una vez tengamos los dispositivos conectados entre sí y el Arduino tenga el código necesario cargado, procederemos a enviar datos y tendremos resultados como los de la figura 2.46.

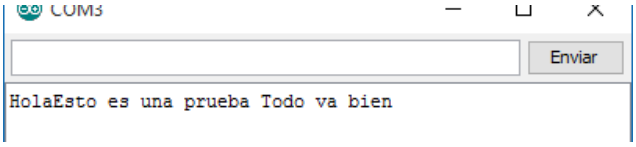

Figura 2.46: Pruebas conexión Bluetooth

Este módulo es una pieza fundamental en este proyecto, pues comunicará el vehículo con la aplicación de móvil. Tendrá la tarea tanto de recibir datos de la aplicación móvil enviados desde el vehículo, como de enviar mediciones de los sensores o estados de éste a la aplicación.

2.5 Otros elementos

En este apartado veremos el resto de componentes de los que disponemos en el proyecto, como los leds o los zumbadores.

2.5.1 Zumbador

El zumbador es un transductor electroacústico (dispositivo que transforma la electricidad en sonido, o viceversa) que produce un sonido o zumbido continuo o intermitente dentro de un mismo tono. Sirve como mecanismo de señalización o aviso, y son utilizados en múltiples sistemas, como en automóviles o en electrodomésticos. Lo podemos ver en la figura 2.47 y el papel en nuestro proyecto es el de simular un claxon de un vehículo que se activará cuando se envíe la orden desde la aplicación móvil.

Figura 2.47: Zumbador

Funcionamiento

Consta de dos elementos, un electroimán y una lámina metálica de acero. Cuando se acciona, la corriente pasa por la bobina del electroimán y produce un campo magnético variable que hace vibrar la lámina de acero sobre la armadura.

Conexión

La conexión es simple: la patilla negativa del zumbador se conecta a tierra del Arduino (GND), y la patilla positiva a un pin del Arduino configurado como salida, como se muestra en la figura 2.48. Se puede utilizar un pin PWM para enviar un pulso o cualquier pin y ponerlo a nivel alto. Se conectará una resistencia entre el pin y el zumbador para evitar sobrecargas.

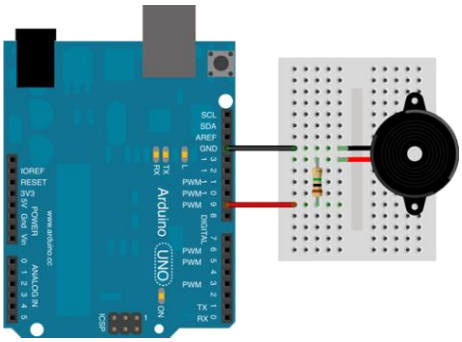

Figura 2.48: Conexión zumbador

Código

El zumbador se puede programar con la función *tone*, que emite un tono por el pin que se le indica, a la frecuencia deseada y de duración programable. Otro método para programarlo es poner la salida a nivel alto o nivel bajo.

```
int pinZumbador = 10;
void setup(){
        pinMode (pinZumbador, OUTPUT);
}
void loop (){
        tone (pinZumbador, 100, 500); //pin, frecuencia y tiempo
        delay (1000); //Esperamos un segundo
        tone (pinZumbador, 1000, 500); //Emitimos otro tono pero de frecuencia diferente
}
```

2.5.2 LEDs

Un LED (Diodo Emisor de Luz, en inglés Light Emitting Diode) es un tipo especial de diodo que trabaja como un diodo común, pero que al ser atravesado por la corriente eléctrica, emite luz (figura 2.49).

Figura 2.49: Diodo LED

Existen diodos LED de varios colores que dependen del material con el que fueron construidos. El diodo LED tiene tres componentes básicos: el encapsulado, el ánodo (positivo) y el cátodo (negativo). La polaridad del diodo LED se puede saber de dos maneras directas:

- La pata más larga siempre es el ánodo.
- En el lado del cátodo, la base del led tiene un borde plano.

En este proyecto vamos a utilizar diodos LED para simular el "semáforo" y las luces del vehículo, haciendo tanto de faros delanteros, que serán de color blanco, como de luces traseras, que serán de color rojo. Las luces blancas del vehículo se encenderán cuando el usuario quiera, mandando la orden desde la aplicación, mientras que las luces rojas se encenderán siempre y cuando al coche se le envíe la instrucción de parar desde la aplicación.

El "semáforo" será controlado por otro Arduino y tendrá un LED rojo encendido hasta que el vehículo esté lo suficientemente cerca y en ese momento se apagará el LED rojo y encenderá un LED verde. Esta distancia la sabremos cuando el sensor de ultrasonidos del que dispone el "semáforo" vea que el vehículo se encuentra a una cierta distancia programada con anterioridad, en nuestro caso de 10cm.

Especificaciones técnicas

- Tensión de trabajo: 1,5-2,2V.
- Corriente de trabajo: 10-20mA.

Funcionamiento

Cuando un LED se encuentra en polarización directa, los electrones pueden recombinarse con los huecos en el dispositivo, liberando energía en forma de fotones. Este efecto es llamado electroluminiscencia, y el color de la luz se determina a partir de la banda de energía del semiconductor.

Conexión

Como el ánodo es el extremo positivo del diodo irá conectado al pin de alimentación del Arduino con una resistencia en serie para evitar que demasiada corriente pueda dañarlo. En la figura 2.50 se muestra la conexión, en la cual no hay resistencia porque el pin 13 del Arduino tiene una resistencia interna ya que el propio Arduino tiene un diodo Led que se controla con este pin.

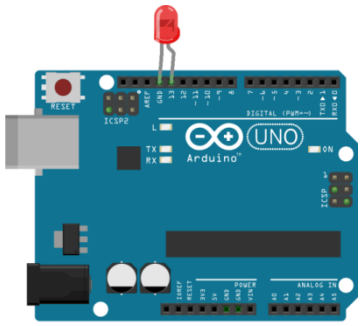

Figura 2.50: Conexionado LED

Código

```
void setup() {
        pinMode (13, OUTPUT); //Inicializamos el pin 13 como salida
}
void loop() {
        digitalWrite (13, HIGH); // Encendemos el Led
        delay (1000); // Esperamos un segundo
        digitalWrite (13, LOW); // Apagamos el Led
        delay (1000);
}
```

Capítulo 3. Implementación software

La parte software del proyecto consta de dos bloques: la programación del vehículo y del "semáforo" en Arduino y la programación de la App en Android. Estos dos bloques se comunicarán entre sí mediante Bluetooth y transmitirán información y órdenes entre ellos.

3.1 Programación del Arduino

Como nuestro vehículo tendrá dos modos de funcionamiento, control manual y control automático, la programación de éste se hará en función de esta conexión. En el diagrama 3.1 se puede ver cómo están organizados estos modos:

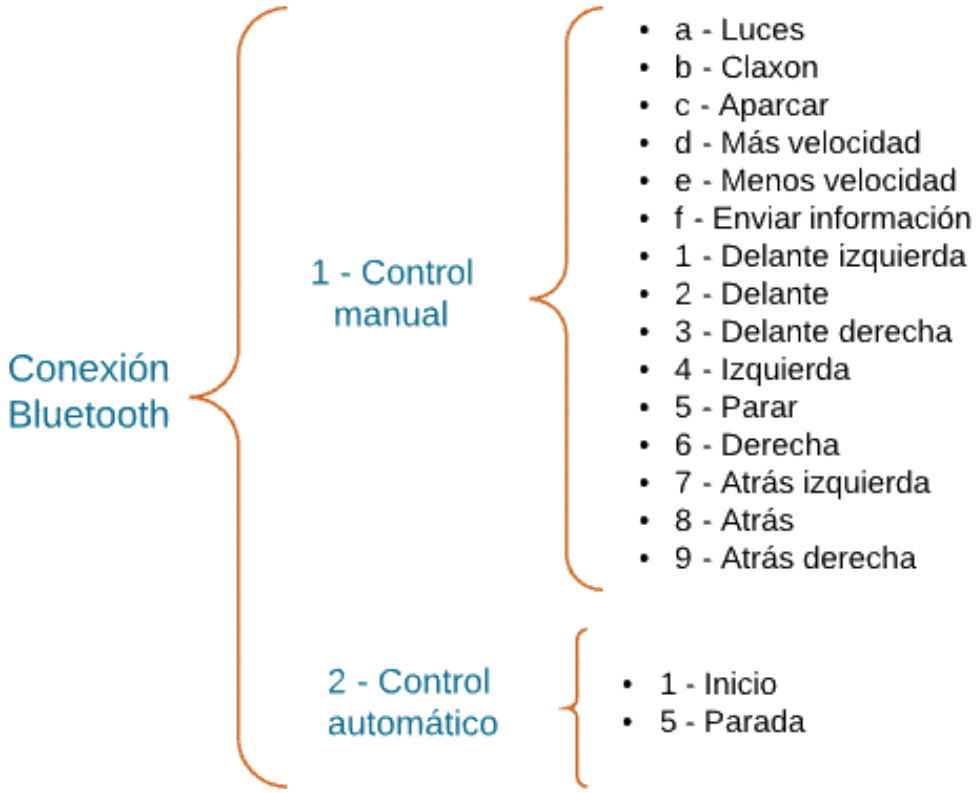

Diagrama 3.1: Códigos de funcionamiento del vehículo

En un primer momento el vehículo se encuentra en modo reposo, esperando recibir instrucciones por parte de la aplicación. La primera instrucción se recibe tras realizar la conexión Bluetooth y es la que le indica qué modo de funcionamiento va a seguir:

- **Control Manual**. El vehículo recibe un '1' y comenzamos a manejarlo de forma manual, lo que nos permite pleno control del vehículo desde la aplicación móvil. Todas las acciones que podemos realizar están contempladas en el diagrama 3.1, en el cual cada acción tiene asociado un valor que se envía desde la aplicación al vehículo; por ejemplo, si recibe un '1' es moverse delante a la izquierda, o si recibe una 'a' encenderá las luces. La instrucción "aparcar" nos detiene el vehículo y termina la conexión Bluetooth, lo que implica terminar el control manual.
- **Control Automático**. El vehículo recibe un '2' y espera a iniciar un movimiento automático, que se activa al pulsar un '1' y se trata de un movimiento autónomo por parte del vehículo en diferentes entornos. Una vez terminado el movimiento automático, podremos desconectarnos cuando se reciba un '5'. El movimiento automático estará en función de los resultados obtenidos con los sensores previamente instalados y programados en el Arduino.

El modo manual está programado con una instrucción *switch/case* para elegir la acción dentro de un bucle *while* y estar siempre realizando acciones mientras no nos desconectemos. El código lo podemos ver en la figura 3.1, donde se observa que las instrucciones están codificadas en formato ASCII y corresponden a las instrucciones establecidas en el diagrama 3.1.

```
while (byteread != 99){ //se envía una c
  while ( bluetooth.available()==0){}
  byteread = bluetooth.read();

  switch (byteread){
    case 49:  delante_izq();
              break;
    case 50:  delante();
              break;
    case 51:  delante_der();
              break;
    case 52:  izquierda();
              break;
    case 53:  parar();
              break;
    case 54:  derecha();
              break;
    case 55:  atras_izq();
              break;
    case 56:  atras();
              break;
    case 57:  atras_der();
              break;
    case 97:  if (controlLuz==0){
                digitalWrite(4, HIGH);
                //digitalWrite(22, HIGH);
                controlLuz=1;
              }
              else{
                digitalWrite(4, LOW);
                //digitalWrite(22, LOW);
                controlLuz=0;
              }
              break;
    case 98:  if (controlZum == 0){
                digitalWrite(5, HIGH);
                controlZum = 1;
              }
              else{
                //Serial.println(" Zumbador OFF");
                digitalWrite(5,LOW);
                controlZum = 0;
              }
              break;
    case 100: VLC_IZQ = VLC_IZQ + 5;
              VLC_DER = VLC_DER - 5;
              if (VLC_IZQ >= 442 && VLC_DER <= 321){
                VLC_IZQ = 442;
                VLC_DER = 321;
              }
              actualizar_vlc();
              break;
    case 101: VLC_IZQ = VLC_IZQ - 5;
              VLC_DER = VLC_DER + 5;
              if (VLC_IZQ <= 390 && VLC_DER >= 372){
                VLC_IZQ = 390;
                VLC_DER = 372;
              }
              actualizar_vlc();
              break;
    case 102: parar();
              delay(500);
              envio_man();
              break;
    default: parar();
   } //end switch

} //end while
```

Figura 3.1: Código para el movimiento manual.

En el modo automático se debe realizar una programación por partes. Como nuestro vehículo va a ir por "entornos", éstos serán programados de manera consecutiva en el código de nuestro vehículo. Para programarlos también utilizaremos la instrucción *switch/case* y una variable "estado", que nos indicará el estado en el que se encuentra el vehículo en función de la acción que esté realizando.

Cuando el vehículo empiece a moverse evitando líneas negras será el "estado 1", siguiendo en él hasta llegar a una franja negra perpendicular a las líneas, donde se detendrá para configurar el sensor TCS230 que mide los colores en el suelo, "estado 2". Aquí comienza el segundo entorno. Una vez configurado, seguiremos evitando líneas y deteniéndose cada vez que haya una franja negra para ver si está el "semáforo" o realizar mediciones en el suelo y obtener el color, "estado 3". Continuaremos hasta encontrarnos con el "semáforo", donde se esperará a que éste se ponga verde y proseguir evitando obstáculos, "estado 4" y tercer entorno de nuestro circuito. Finalmente, cuando el vehículo no sea capaz de proseguir su camino porque los obstáculos le rodean, pasará al "estado 5" y finalizará el movimiento automático.

Cada vez que se cambia de estado o se realiza una medida de color se envía la información a la aplicación del móvil, teniendo así toda la información en el dispositivo.

En la figura 3.2 se muestra el código de seguimiento de líneas negras que se ha utilizado para que el vehículo las evite y se pare cuando haya una franja, y en la figura 3.3 se muestra la función que rectifica el movimiento del vehículo en función de la cantidad de línea negra detectada.

```
switch (estado){
  case 1: //envio de informacion
          bluetooth.write("#");
          bluetooth.write("2"); //texto en la app: segir lineas
          bluetooth.write("~");
          delay(500);
          while (estado != 2){
            moverse();
            linea_izq = analogRead(A0);
            linea_der = analogRead(A1);
            //en caso de que ambos detecten linea negra, se para
            if ( (linea_izq < 600) && (linea_der < 600) ) {
              seguidor (linea_izq, linea_der);
            }
          else {
            parar();
            //enviamos info
            bluetooth.write("#");
            bluetooth.write("3"); //cambiamos de estado
            bluetooth.write("~");
            delay(500);
            estado = 2;
          }
        }
        break;
```

Figura 3.2: código movimiento por línea negra.

```
void seguidor (int linea_izq, int linea_der) {
  if (linea_der > 600){
    VLC_DER = VLC_DER + 5;
    VLC_IZQ = VLC_IZQ + 5;
    moverse();
    delay(200);
    VLC_DER = VLC_DER - 5;
    VLC_IZQ = VLC_IZQ - 5;
    moverse();
  }
  if ((100 < linea_der) && (linea_der < 600)){
    VLC_DER = VLC_DER + 3;
    VLC_IZQ = VLC_IZQ + 3;
    moverse();
    delay(100);
    VLC_DER = VLC_DER - 3;
    VLC_IZQ = VLC_IZQ - 3;
    moverse();
  }
  if (linea_izq > 600){
    VLC_IZQ = VLC_IZQ - 5;
    VLC_DER = VLC_DER - 5;
    moverse();
    delay(200);
    VLC_IZQ = VLC_IZQ + 5;
    VLC_DER = VLC_DER + 5;
    moverse();
  }
  if ((100 < linea_izq) && (linea_izq < 600)){
    VLC_IZQ = VLC_IZQ - 3;
    VLC_DER = VLC_DER - 3;
    moverse();
    delay(100);
    VLC_IZQ = VLC_IZQ + 3;
    VLC_DER = VLC_DER + 3;
    moverse();
  }
}
```

Figura 3.3: código función rectificación del movimiento.

Los códigos utilizados para programar cada componente se pueden ver por separado en los apartados correspondientes de cada elemento hardware descritos en el capítulo 2. Por otro lado, la programación del semáforo se basa en detectar una distancia menor de 10cm mediante un sensor de ultrasonidos y, cuando se cumpla, apagar el LED rojo y encender el verde.

3.2 Programación de la aplicación en Android

Una aplicación móvil o App es una aplicación informática diseñada para su uso en un teléfono inteligente, tabletas y otros dispositivos móviles. Nuestra App tiene funcionalidad básica, de fácil construcción y uso. Como es para un uso concreto, la App será una aplicación nativa, esto es, nos basaremos en un entorno de desarrollo, Android, ya que es el más sencillo y del que disponemos de más información y facilidades.

Una aplicación Android puede contener uno o más elementos:

- Activities (actividades): elemento básico de interacción con el usuario, equivalente a una ventana o caja de diálogo en una aplicación convencional de ordenador.
- Content providers (proveedores de contenido): acceso de datos por parte de cualquier aplicación del sistema.
- Services (servicios): aplicación sin interacción directa con usuario y con ciclo de vida más largo.
- Intents (propósitos o intenciones): mensajes asíncronos que permiten que unos componentes de Android utilicen otros componentes.

Las actividades controlan el ciclo de vida de una aplicación, dado que el usuario no cambia de aplicación, sino de actividad. El sistema mantiene una pila con las actividades previamente visualizadas, de forma que el usuario puede regresar a la actividad anterior pulsando la tecla "retorno". El resto de elementos no tienen ciclos de vida, ya que son acciones puntuales.

Una aplicación Android se ejecuta dentro de su propio proceso Linux. Este proceso se crea con la aplicación y continuará en activo hasta que ya no sea requerido y el sistema reclame su memoria para asignársela a otra aplicación.

Una característica importante de Android es que la destrucción de un proceso no es controlada directamente por la aplicación, sino que es el sistema el que determina cuándo destruir el proceso. Lo hace basándose tanto en el conocimiento que tiene de las partes de la aplicación que se están ejecutando (actividades y servicios), como en la importancia de dichas partes para el usuario y en cuánta memoria disponible haya en un determinado momento.

Si tras eliminar el proceso de una aplicación, el usuario vuelve a ella, se crea de nuevo el proceso, pero se habrá perdido el estado que tenía esa aplicación. En estos casos, será responsabilidad del programador almacenar el estado de las actividades, si queremos que cuando sean reiniciadas conserven su estado.

Las actividades pueden estar en tres estados diferentes y pasar de uno a otro mediante llamadas que realiza el sistema al método del siguiente ciclo de vida. Los estados son los siguientes:

- Activa (Resumed): la actividad fue arrancada por el usuario, se ve, se está ejecutando y permite la interacción del usuario.
- Pausada (Paused): la actividad fue arrancada y se está ejecutando, pero no se puede interactuar porque es parcialmente visible (ej.: solapamiento por una notificación).
- Parada (Stopped): la actividad fue arrancada y se está ejecutando, pero está oculta por otras actividades y no permite la interacción con el usuario. Puede haber comunicación por notificaciones.

Existen dos estados transitorios: Creada (Created) y Arrancada (Started), los cuales son movidos por el sistema rápidamente llamando al siguiente estado. También existe otro estado en el cual la actividad nunca fue arrancada o se finalizó por falta de recursos: Muerta (Destroyed).

Los métodos que hacen que las actividades pasen de un ciclo de vida a otro son:

- onCreate(): se llama cuando se arranca por primera vez la actividad, la actividad estaba parada o se cambian los recursos (de vertical a apaisado).
- onDestroy(): se llama para terminar una actividad.
- onStart(): paso previo a poder interactuar, bien desde su creación o desde su retorno del estado "parado".
- onRestart(): vuelve tras haber estado parada.
- onStop(): paso previo al estado parado.
- onResume(): vuelve a ser visible e interactuable (refresca la interfaz de usuario).
- onPause(): la actividad deja de ser completamente visible e interactuable.

En la figura 3.4 se puede ver el ciclo de vida de una actividad con los métodos que llevan de un estado a otro.

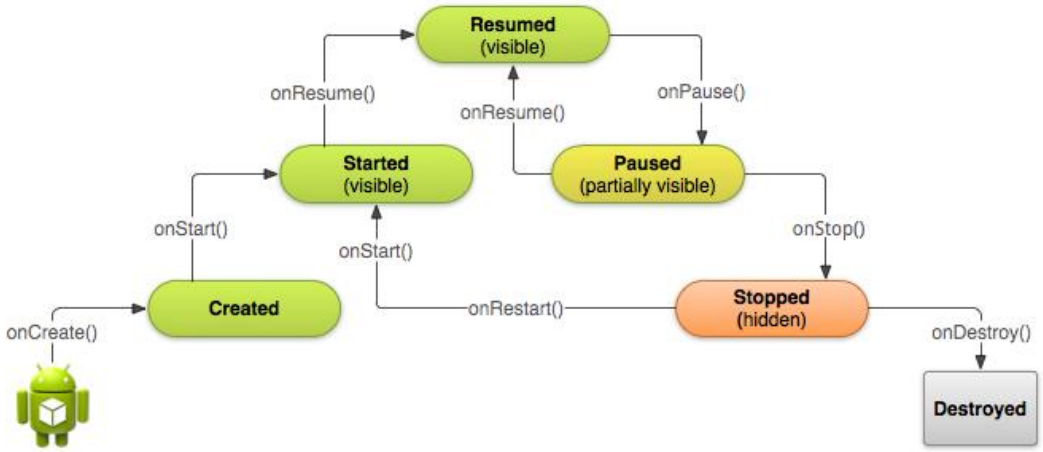

Figura 3.4: Ciclo de vida de una actividad.

Para nuestro proyecto, se ha decidido crear una aplicación móvil para poder manejar y controlar nuestro vehículo. La aplicación constará de tres apartados:

- Comenzar: realiza la conexión Bluetooth entre el vehículo y la App, permitiendo después un modo de funcionamiento u otro.
- Información: se nos muestra una pantalla con todos los componentes del vehículo y una pequeña descripción de cada uno.
- Acerca de: se nos muestra una pantalla con la información sobre el TFG.

La pantalla para el control manual mostrará todas las direcciones en las que pueda ir el vehículo, además de controles como hacer mediciones de temperatura y humedad, encender y apagar las luces o sonar el claxon. Por otro lado, la pantalla de control automático permite iniciar o salir del control automático y mostrará un texto que se actualiza en función del estado del vehículo.

3.2.1 Proceso de creación

Esta es una de las dos ideas principales del TFG: crear una aplicación que permita recibir y enviar instrucciones y datos para tener el control del vehículo y para ello se creará una interfaz sencilla y muy orientativa para facilitar el uso de la App.

Como se ha comentado anteriormente, nuestra aplicación será desarrollada en la plataforma de Android y para ello utilizaremos el programa *Android Studio*, orientado a la creación de apps en lenguaje Android.

El primer paso será crear los bocetos de la aplicación. En una hoja en blanco hemos realizado unos primeros diseños orientativos sobre cómo queremos que sean las interfaces de nuestra aplicación y un flujo básico de funcionamiento entre las mismas. También diseñamos un icono identificativo para la App y que esté relacionado con el trabajo.

La aplicación comienza mostrando en la pantalla del dispositivo móvil el vehículo y de ahí pasa al menú principal donde tendremos los tres apartados comentados anteriormente:

- Comenzar: nos llevará al control del vehículo. Éste será manual o automático, y antes de empezar a controlarlo, nos pedirá que nos conectemos al vehículo mediante Bluetooth. Una vez conectados nos llevará a la pantalla de control que hayamos elegido.
- Información: nos llevará a una pantalla con todos los componentes utilizados en el proyecto y al seleccionar uno se nos abrirá una pantalla con una breve descripción de este.
- Acerca de: nos llevará a una pantalla con una breve descripción del proyecto.

En el diagrama 3.2 podemos el este flujo comentado entre las pantallas.

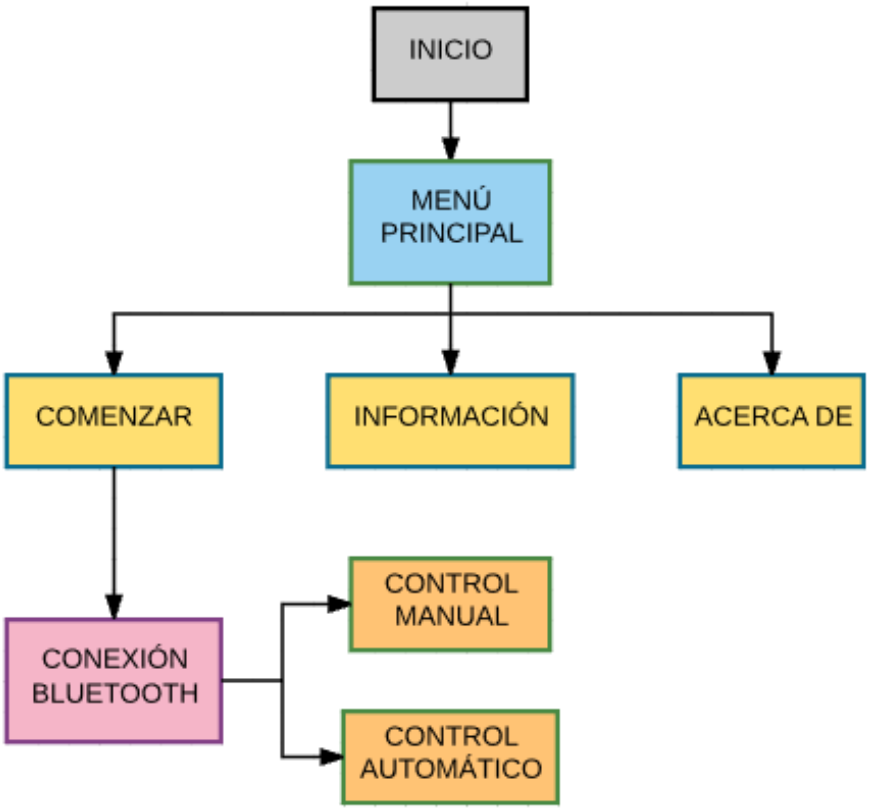

Diagrama 3.2: Diagrama de funcionamiento de la aplicación.

Tras los bocetos y el diagrama de flujo, pasamos a la creación de la aplicación bajo el entorno de desarrollo *Android Studio*.

Android Studio

Esta herramienta es un entorno de desarrollo integrado para la plataforma Android. Su interfaz se puede observar en la figura 3.5, donde aparecen sus apartados más importantes:

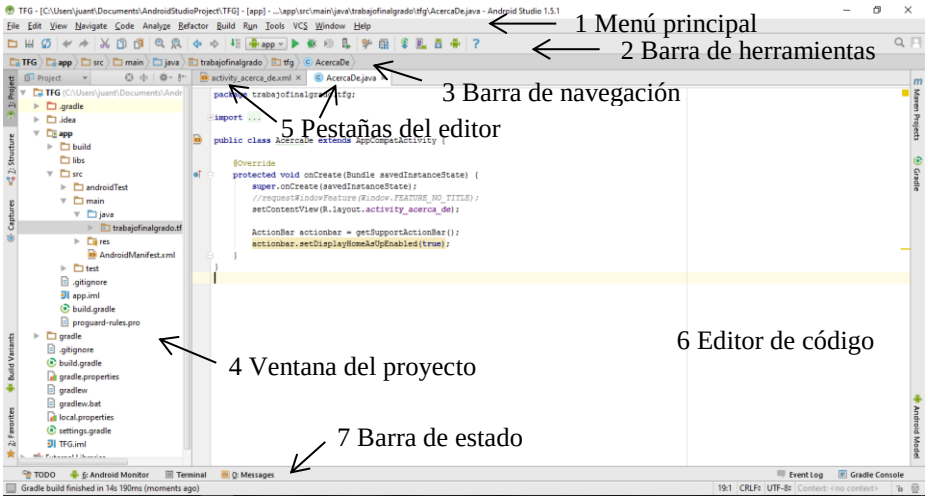

Figura 3.5: Interfaz de Android Studio.

A la hora de crear una pantalla, ésta consta de dos partes: un archivo *.xml*, en el que se crea el diseño de la pantalla (botones, cuadros de texto, imágenes, etc.), y un archivo *.java*, en el que se controlan las acciones que realizan los botones, textos, etc. colocados en el archivo *.xml*.

La interfaz para el diseño varía en función de si queremos realizarlo mediante programación o bien colocar los componentes manualmente. En la figura 3.6 observamos la interfaz en la que se incluyen los componentes a mano. A la izquierda de la figura tenemos un apartado con componentes que se "pinchan" y arrastran para colocarlos donde se quiera en la pantalla del dispositivo; en el centro se muestra un dispositivo móvil que simula a uno real, en el cual se comprueban los resultados al insertar los componentes; en la esquina superior derecha hay un apartado donde se muestran los componentes ya colocados en la pantalla en forma de árbol desplegable, y en la inferior derecha aparece otro apartado en el que se configuran las propiedades para cada componente.

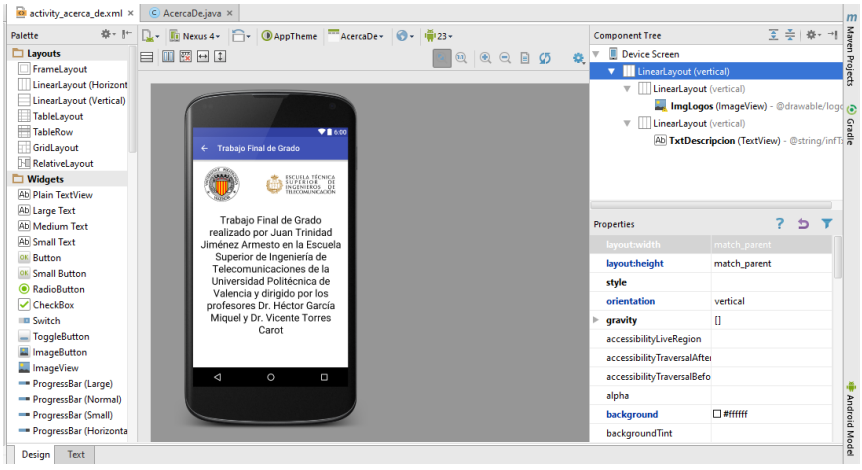

Figura 3.6: Interfaz de diseño a mano.

Se ha de resaltar esta cómoda forma de diseñar, ya que al colocar cada componente se va añadiendo automáticamente el código correspondiente al elemento colocado, generándose así el archivo *.xml*.

Por otro lado, en la figura 3.7 podemos ver la interfaz donde se programa a mano. Dispone de un apartado donde programar y otro para previsualizar en el dispositivo virtual lo que se ha programado. Las características, formas, colores, etc., se introducen mediante código, a diferencia de la otra interfaz que dispone de un apartado en el que realizar la configuración.

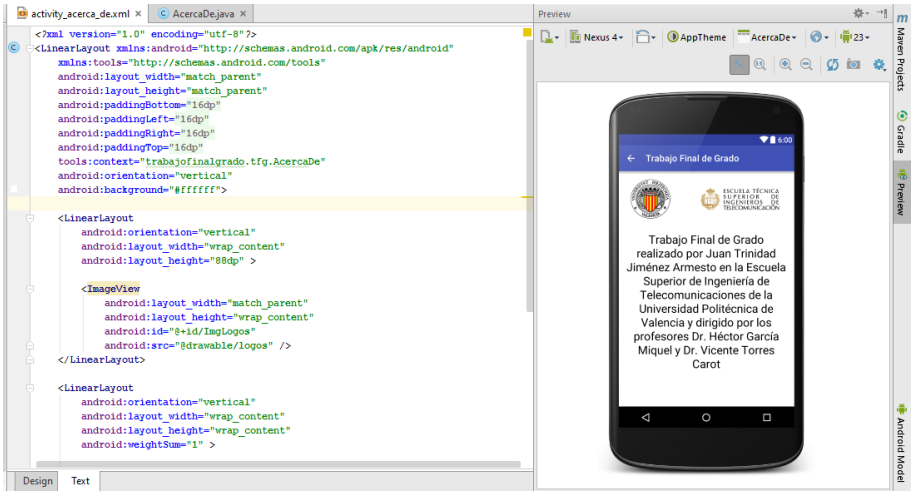

Figura 3.7: Interfaz de diseño mediante programación.

La programación de estos archivos se realiza en lenguaje XML, que se organiza en un elemento raíz del que cuelgan diferentes elementos, que a su vez están formados por atributos y sus valores.

En la figura 3.7 podemos observar que en la segunda línea del código aparece "*<LinearLayout*", que es el elemento raíz, y de él cuelgan sus atributos y valores como "*android: layout_width = "match_parent"*". Si seguimos bajando por las líneas del código, encontramos de nuevo "*<LinearLayout*"; esto es un elemento dentro del elemento raíz y vemos que también tiene sus atributos y valores. Esta organización de los elementos uno dentro de otro construye la estructura en forma de árbol ya comentada que pudimos comprobar en la figura 3.6.

La interfaz de programación de los archivos *.java* aparece en la figura 3.8. En ella vemos que únicamente se dispone de espacio para introducir código en lenguaje JAVA.

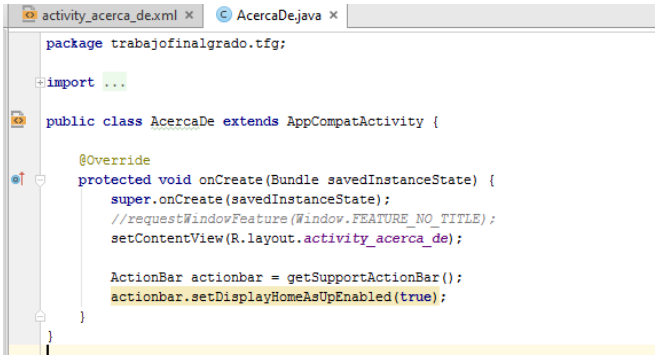

Figura 3.8: Interfaz de programación de archivos .java.

En el archivo *.java* tenemos que relacionar las acciones que queremos realizar con los componentes del diseño, y para ello utilizamos un identificador. Cada elemento del diseño tiene un identificador único, lo que nos permite saber con qué elemento estamos trabajando.

En la figura 3.9 podemos observar cómo creamos las variables de los elementos (1), incluimos en el archivo *.java* el diseño contenido en el archivo *.xml* correspondiente (2), relacionamos las variables creadas con los elementos del diseño mediante los identificadores (3), para después darles una acción (4).

```
Button comenzar, informacion, acercade;  <- 1

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    //requestWindowFeature(Window.FEATURE_NO_TITLE); //oculta el titulo
    setContentView(R.layout.activity_menu_principal);  <- 2

    comenzar = (Button)findViewById(R.id.mp_btn1);
    informacion = (Button)findViewById(R.id.mp_btn2);  <- 3
    acercade = (Button)findViewById(R.id.mp_btn3);

    comenzar.setOnClickListener(new View.OnClickListener() {  <- 4
        @Override
        public void onClick(View v) {
            Intent pulsar = new Intent(MenuPrincipal.this, Comienzo.class);
            startActivity(pulsar);
        }
    });

    informacion.setOnClickListener((v) -> {
            Intent pulsar = new Intent(MenuPrincipal.this, Informacion.class);
            startActivity(pulsar);
    });

    acercade.setOnClickListener((v) -> {
            Intent pulsar = new Intent(MenuPrincipal.this, AcercaDe.class);
            startActivity(pulsar);
    });
```

Figura 3.9: Código de la pantalla principal.

3.2.2 Funcionalidad

Como nuestro deseo es que las pantallas sean lo más intuitivas, sencillas y orientativas posible, le añadiremos botones, textos e imágenes que resulten familiares para el usuario/a y que representen la acción que se desea ejecutar o que muestren una determinada información.

Una vez finalizado el proceso de creación, se procederá a testearla con el simulador que posee *Android Studio* para ver el funcionamiento y corregir posibles errores y problemas que aparezcan, además de incluir posibles mejoras que se consideren oportunas.

Posteriormente pasaremos a utilizar la aplicación en nuestro dispositivo móvil. Al cargar la App es posible que todavía haya que modificar o corregir algún detalle ya que el cambio de dispositivo hace que las pantallas tengan distribuciones diferentes. Una vez corregidos y comprobado que la aplicación funciona correctamente, comenzamos a trabajar con ella. En caso de posibles fallos durante el funcionamiento, siempre se está a tiempo de corregir y mejorar.

Como el código es muy extenso, sólo veremos la parte que interactúa con el vehículo, es decir, la interpretación de las instrucciones que recibe y las que envían. En primer lugar veremos el control manual y más adelante el automático.

Control manual

La interfaz para el control manual del vehículo la podemos ver en la figura 3.10. Consiste en ocho botones que indican las direcciones de desplazamiento, otro que lo detiene, otros dos manejan la velocidad, otro controla las luces, otro más lo hace con el claxon y por fin otro llamado “Aparcar” que detiene el vehículo y finaliza su conexión con la aplicación. Además, se ha incluido un botón que nos actualiza la información que nos proporcionan los sensores del vehículo: la temperatura ambiental del momento, la humedad y la distancia desde el vehículo hasta el objeto más cercano al mismo.

Figura 3.10: Interfaz para el control manual.

Cada uno de estos botones lleva asociado un valor que se envía al vehículo, especificados en el diagrama 3.1. El código de Android para este control manual lo podemos comprobar en la figura 3.11 y consiste en que cuando se pulse un botón determinado se envíe la instrucción correspondiente.

```
public void onClick(View v) {

    switch (v.getId())
    {
        case(R.id.ManualArIz):
            mConnectedThread.write("1");
            break;
        case(R.id.ManualAr):
            mConnectedThread.write("2");
            break;
        case(R.id.ManualArDe):
            mConnectedThread.write("3");
            break;
        case(R.id.ManualIz):
            mConnectedThread.write("4");
            break;
        case(R.id.ManualStop):
            mConnectedThread.write("5");
            break;
        case(R.id.ManualDe):
            mConnectedThread.write("6");
            break;
        case(R.id.ManualAbIz):
            mConnectedThread.write("7");
            break;
        case(R.id.ManualAb):
            mConnectedThread.write("8");
            break;
        case(R.id.ManualAbDe):
            mConnectedThread.write("9");
            break;
        case(R.id.ManualBtnL):
            mConnectedThread.write("a");
            break;
        case(R.id.ManualBtnC):
            mConnectedThread.write("b");
            break;
        case(R.id.ManualBtnA):
            mConnectedThread.write("c");
            Intent pulsar = new Intent(MovManual.this, Comienzo.class);
            startActivity(pulsar);
            break;
        case(R.id.ManualMasVlc):
            mConnectedThread.write("d");
            break;
        case(R.id.ManualMenVlc):
            mConnectedThread.write("e");
            break;
        case(R.id.ManActBtn):
            mConnectedThread.write("f");
            break;
    }
}
```

Figura 3.11: Código Android para el control manual.

Como la aplicación también tiene que recibir información de los sensores, se va a dividir la recepción en dos partes: la velocidad y los sensores. Como podemos ver en la figura 3.12, comprobamos qué valor tiene el primer carácter recibido y en función de ello actualizamos la velocidad, cuando recibe un 1 primero, o el valor de las mediciones de los sensores, si recibe un 2.

```
if (dataInPrint.charAt(0) == '1'){
    String dato = dataInPrint.substring(1, endOfLineIndex-1);
    txtVlc.setText(dato + " m/s");
}

if (dataInPrint.charAt(0) == '2'){
    int j=0;
    int [] pos = new int[3];
    int dataLength = dataInPrint.length();
    for (int i=0; i < dataLength; i++){
        if(dataInPrint.charAt(i) == '/'){
            pos[j]=i;
            j++;
        }
    }
    String dato1 = dataInPrint.substring(1, pos[0]);
    String dato2 = dataInPrint.substring(pos[0]+1, pos[1]);
    String dato3 = dataInPrint.substring(pos[1]+1, pos[2]);

    txtTemp.setText(dato1 + " °C");
    txtHum.setText(dato2 + " %");
    if (dato3 == "100")
        txtDis.setText(">" + dato3 + " cm");
    else
        txtDis.setText(dato3 + " cm");
```

Figura 3.12: Código reconocimiento de datos en control manual

Control automático

La parte automática es más sencilla en cuanto a la interfaz, pero más compleja en lo referente a la recepción de datos y envío de instrucciones, ya que la aplicación está constantemente recibiendo datos de la situación del vehículo. Como podemos ver en la figura 3.13, la pantalla del control automático consta de dos botones, uno para iniciar el movimiento y otro para terminar la conexión entre el vehículo y la aplicación cuando haya terminado de moverse. La pantalla dispone de dos textos, en uno de ellos se informa del funcionamiento del vehículo y en qué estado se encuentra (texto “información”); y en el otro se muestran las mediciones de los colores (texto “R G B”). También se realiza una medición de la temperatura y humedad, que se muestran en los textos de la parte inferior de la pantalla.

Figura 3.13: Interfaz para el control automático

La información que se recibe en este control automático se elabora y se detecta y se trata con los mismos criterios que en movimiento manual. En la figura 3.14 podemos ver los diferentes estados del vehículo y las instrucciones para mostrarlas por pantalla.

```
if (dataInPrint.charAt(0) == '1'){
    String datoTemp = dataInPrint.substring(1, 6);
    String datoHum = dataInPrint.substring(6, endOfLineIndex-1);
    txtR.setVisibility(View.INVISIBLE);
    txtG.setVisibility(View.INVISIBLE);
    txtB.setVisibility(View.INVISIBLE);
    txtInfo.setText("Comienza");
    txtTemp.setText(datoTemp + " °C");
    txtHum.setText(datoHum + " %");
}

if (dataInPrint.charAt(0) == '2'){
    txtR.setVisibility(View.INVISIBLE);
    txtG.setVisibility(View.INVISIBLE);
    txtB.setVisibility(View.INVISIBLE);
    txtInfo.setText("Siguiendo lineas");
}

if (dataInPrint.charAt(0) == '3'){
    txtInfo.setText("Cambio de estado");
    txtR.setVisibility(View.INVISIBLE);
    txtG.setVisibility(View.INVISIBLE);
    txtB.setVisibility(View.INVISIBLE);
}

if (dataInPrint.charAt(0) == '4'){
    txtInfo.setText("Calibramos sensor TCS");
}

if (dataInPrint.charAt(0) == '8'){
    int j=0;
    int [] pos = new int[3];
    int dataLength = dataInPrint.length();
    for (int i=0; i < dataLength; i++){
        if(dataInPrint.charAt(i) == '/'){
            pos[j]=i;
            j++;
        }
    }
    String colorR = dataInPrint.substring(1, pos[0]);
    String colorG = dataInPrint.substring(pos[0]+1, pos[1]);
    String colorB = dataInPrint.substring(pos[1]+1, pos[2]);
    txtR.setVisibility(View.VISIBLE);
    txtG.setVisibility(View.VISIBLE);
    txtB.setVisibility(View.VISIBLE);
    txtInfo.setText("Color suelo");
    txtR.setText(colorR);
    txtG.setText(colorG);
    txtB.setText(colorB);
}
```

```
if (dataInPrint.charAt(0) == '5'){
    int j=0;
    int [] pos = new int[3];
    int dataLength = dataInPrint.length();
    for (int i=0; i < dataLength; i++){
        if(dataInPrint.charAt(i) == '/'){
            pos[j]=i;
            j++;
        }
    }
    String colorR = dataInPrint.substring(1, pos[0]);
    String colorG = dataInPrint.substring(pos[0]+1, pos[1]);
    String colorB = dataInPrint.substring(pos[1]+1, pos[2]);
    txtR.setVisibility(View.VISIBLE);
    txtG.setVisibility(View.VISIBLE);
    txtB.setVisibility(View.VISIBLE);
    txtInfo.setText("Color suelo");
    txtR.setText(colorR);
    txtG.setText(colorG);
    txtB.setText(colorB);
}
if (dataInPrint.charAt(0) == '6'){
    txtR.setVisibility(View.INVISIBLE);
    txtG.setVisibility(View.INVISIBLE);
    txtB.setVisibility(View.INVISIBLE);
    txtInfo.setText("Nos paramos para medir distancia");
}

if (dataInPrint.charAt(0) == '7'){
    txtInfo.setText("Esperamos al semáforo");
    txtR.setVisibility(View.INVISIBLE);
    txtG.setVisibility(View.INVISIBLE);
    txtB.setVisibility(View.INVISIBLE);
i
}
    txtG.setVisibility(View.INVISIBLE);
    txtB.setVisibility(View.INVISIBLE);
    txtInfo.setText("Movimiento evitando obstáculos");
}

if (dataInPrint.charAt(0) == 'a'){
    txtInfo.setText("Objeto delante, recalculando...");
}

if (dataInPrint.charAt(0) == 'b'){
    txtInfo.setText("Volvemos a movernos evitando obstáculos");
}

if (dataInPrint.charAt(0) == 'c'){
    txtInfo.setText("Modo automático terminado, pulse STOP para salir");
}
```

Figura 3.14: Código reconocimiento de datos en control automático

Tras las pertinentes pruebas, la aplicación consigue recibir información desde el vehículo, la elabora y la muestra en la pantalla del dispositivo; posteriormente le envía al vehículo otra información para que sea tratada por él y realice las oportunas operaciones

3.3 Comunicación

Como se indicó anteriormente, la comunicación entre el vehículo y la aplicación se realiza mediante Bluetooth, por lo que tendremos dos códigos diferentes, el de Arduino para el vehículo y el de Android para la aplicación.

En el apartado 2.4 comentamos el código Arduino, pero en esta ocasión resaltaremos que cuando en el vehículo se reciba algún dato, el sistema comprobará a qué se refiere y realizará la

acción que tuviera vinculada ese dato. Esto lo podemos ver perfectamente en la figura 3.1 del apartado 3.1, donde se puede comprobar cómo el vehículo queda a la espera hasta recibir un dato y tras ello realiza la acción correspondiente.

El código de Android para la aplicación es más complejo, ya que la App es la que inicia la comunicación con el módulo. Desde la pantalla de comienzo, seleccionaremos uno de los dos modos y entraremos a la pantalla de conexión, donde tendremos una serie de fases para realizar la conexión:

1. Comprobar que el dispositivo móvil posee Bluetooth. En caso de disponer de él, ver si está disponible y si no es así, activarlo (figura 3.15).

```
private void checkBTState() {
    // Check device has Bluetooth and that it is turned on
    mBtAdapter=BluetoothAdapter.getDefaultAdapter(); // CHECK THIS OUT THAT IT WORKS!!!
    if(mBtAdapter==null) {
        Toast.makeText(getBaseContext(), "El dispositivo no soporta Bluetooth", Toast.LENGTH_SHORT).show();
    } else {
        if (mBtAdapter.isEnabled()) {
            Log.d(TAG, "...Bluetooth Activado...");
        } else {
            //Prompt user to turn on Bluetooth
            Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, 1);

        }
    }
}
```

Figura 3.15: Comprobación del estado del Bluetooth.

2. Una vez activado hay que buscar los elementos que ya tengamos enlazados con nuestro dispositivo y mostrarlos en una lista para seleccionar directamente nuestro módulo, que ya tendremos vinculado con el dispositivo móvil (figura 3.16).

```
// Initialize array adapter for paired devices
mPairedDevicesArrayAdapter = new ArrayAdapter<String>(this, R.layout.device_name);

// Find and set up the ListView for paired devices
ListView pairedListView = (ListView) findViewById(R.id.paired_devices);
pairedListView.setAdapter(mPairedDevicesArrayAdapter);
pairedListView.setOnItemClickListener(mDeviceClickListener);

// Get the local Bluetooth adapter
mBtAdapter = BluetoothAdapter.getDefaultAdapter();

// Get a set of currently paired devices and append to 'pairedDevices'
Set<BluetoothDevice> pairedDevices = mBtAdapter.getBondedDevices();

// Add previosuly paired devices to the array
if (pairedDevices.size() > 0) {
    findViewById(R.id.title_paired_devices).setVisibility(View.VISIBLE);//make title viewable
    for (BluetoothDevice device : pairedDevices) {
        mPairedDevicesArrayAdapter.add(device.getName() + "\n" + device.getAddress());
    }
} else {
    String noDevices = "Ningun dispositivo pudo ser emparejado".toString();
    mPairedDevicesArrayAdapter.add(noDevices);
}
```

Figura 3.16: Listado de elementos vinculados al dispositivo móvil.

3. Conectarnos con el dispositivo que tendremos en la lista anteriormente comentada (figura 3.17). En caso de error en el momento de la conexión, nos avisará de que ha habido un problema y no se ha podido conectar.

```
private AdapterView.OnItemClickListener mDeviceClickListener = new AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> av, View v, int arg2, long arg3) {

        textView1.setText("Conectando");
        // Get the device MAC address, which is the last 17 chars in the View
        String info = ((TextView) v).getText().toString();
        String address = info.substring(info.length() - 17);

        // Make an intent to start next activity while taking an extra which is the MAC address.
        Intent i = new Intent(ConectarAutomatico.this, MovAutomatico.class);
        i.putExtra(EXTRA_DEVICE_ADDRESS, address);
        startActivity(i);
    }
};
```

Figura 3.17: Conexión con el dispositivo.

4. Enviar y recibir datos tras la conexión. El envío de datos se hará en el momento de pulsar un botón, como vemos en la figura 3.18, mientras que la recepción se realiza constantemente (figura 3.19).

```
play.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        mConnectedThread.write("1");    // Send "1" via Bluetooth
    }
});
```

Figura 3.18: Envío de datos.

```
bluetoothIn = new Handler() {
    public void handleMessage(android.os.Message msg) {
        if (msg.what == handlerState) {
            String readMessage = (String) msg.obj;
            recDataString.append(readMessage);
```

Figura 3.19: Recepción de datos.

En la figura 3.19 se observa cómo se reciben los datos y se convierten a un String, en donde posteriormente se operará tal y como se vio en las figuras 3.12 y 3.14 del apartado 3.2.1.

Capítulo 4. Estudio de la funcionalidad y de los resultados obtenidos

Tras el montaje del vehículo y la programación tanto de éste como de la aplicación, pasamos a comprobar la funcionalidad global para estudiar su comportamiento.

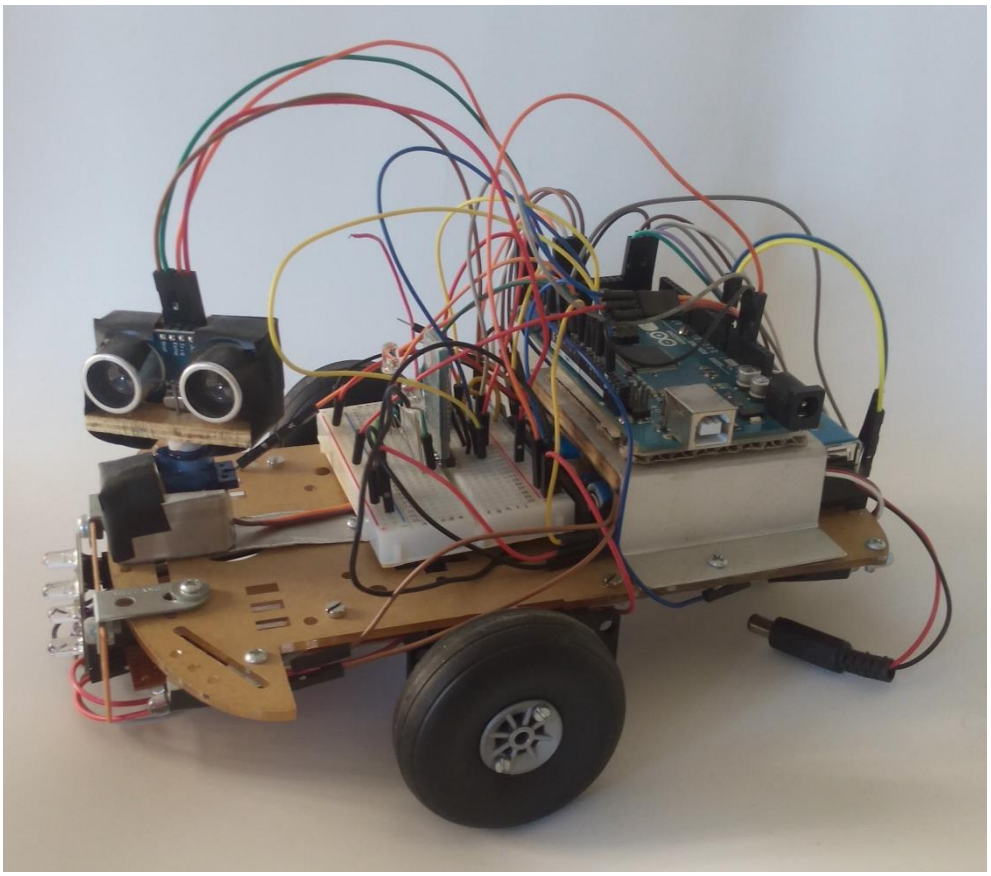

Figura 4.1: Vehículo ensamblado.

En primer lugar, podemos ver el vehículo con todos sus componentes en la figura 4.1. Su montaje fue laborioso y no exento de problemas. El más importante se centró en el funcionamiento de las ruedas, pues iban desfasadas ante la misma tensión de entrada. Para solucionarlo se cambió el tipo de motor, pasando de utilizar un motor de corriente continua a motores servo. Tuvimos también un problema con las baterías, ya que el vehículo utiliza tres motores servo y no había suficiente alimentación, por lo que se tuvo que aumentar la capacidad de alimentación del conjunto poniendo dos baterías de 9V. Otro problema más leve se debió a que, como se dispone de muchos elementos, se necesitaban muchas piezas para sujetarlos, por lo que algunas

se crearon *ex proceso* para seguir con el montaje y otras se utilizaron del clásico juego "Mecano".

Por otro lado, en la figura 4.2 podemos ver cómo queda finalmente nuestra aplicación.

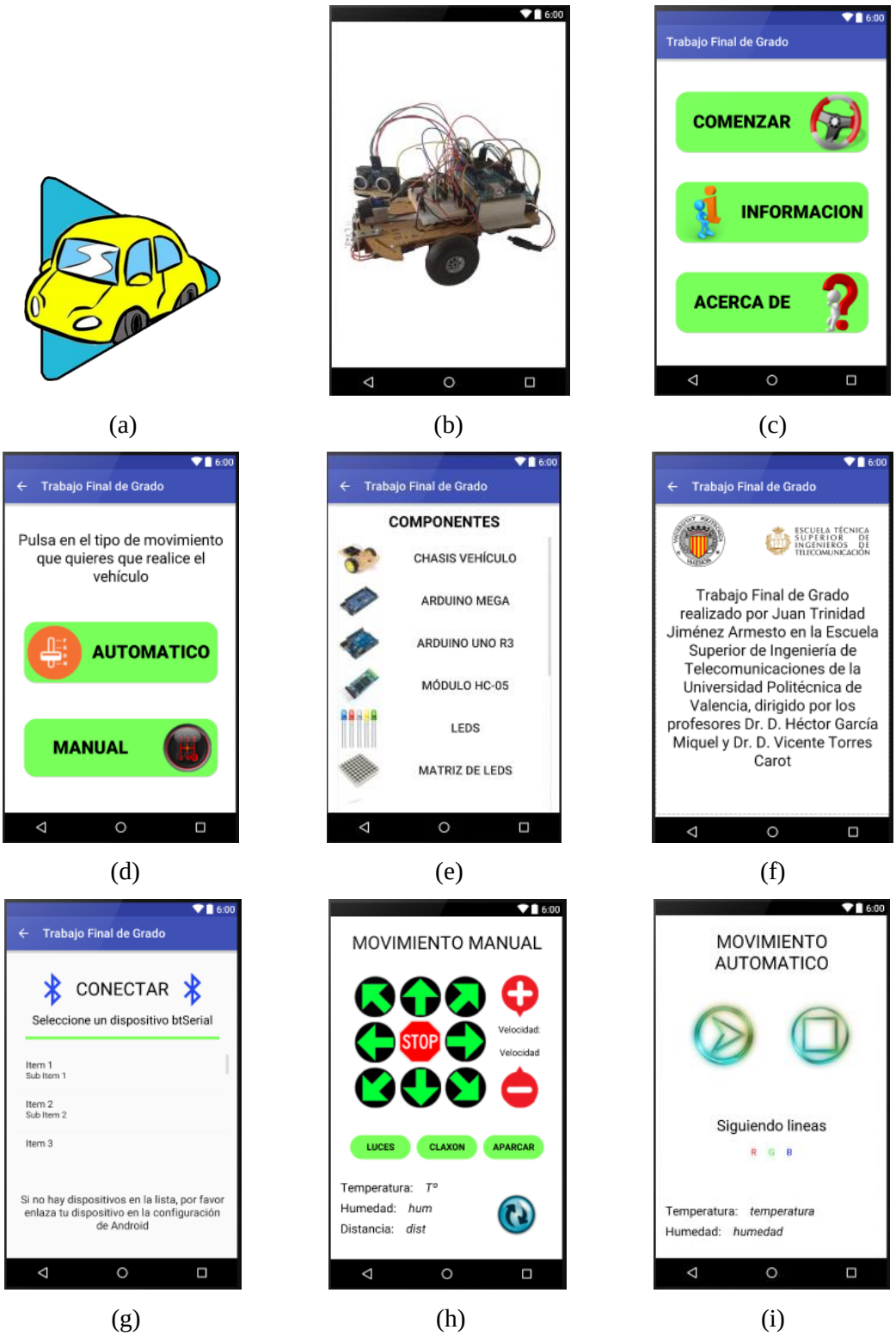

Figura 4.2 Diferentes pantallas de la App: (a) Icono, (b) Pantalla de inicio, (c) Menú principal, (d) Pantalla de comenzar, (e) Pantalla de información, (f) Pantalla de "acerca de", (g) Pantalla de conexión Bluetooth, (h) Pantalla de control manual, (i) Pantalla de control automático.

Una vez cumplidos los objetivos de este TFG, pasamos a comentar los resultados obtenidos relacionados con el funcionamiento del vehículo y la aplicación. Con la intención de comprobar los diferentes modos de funcionamiento, se realizó un código para Arduino con el cual mostrar por el terminal del entorno de desarrollo de Arduino que se había entrado en un modo u otro. En las figuras 4.3 y 4.4 podemos ver cómo se muestra el modo al que se accede, qué instrucción se envía desde la aplicación y qué acción realiza el vehículo.

Desde la aplicación accedemos al modo manual, nos conectamos y se envía el primer dato al vehículo, indicándole que entramos al movimiento manual. En la figura 4.3 vemos que hemos entrado en él y el vehículo espera a recibir una instrucción. Cuando la recibe, lee qué instrucción es y la muestra por el terminal. Este tratamiento de las instrucciones corresponde al visto en el diagrama 3.1 del capítulo 3.

Figura 4.3: Comprobación del modo de funcionamiento manual.

En esta ocasión accedemos al movimiento automático de la misma manera que con el manual. En la figura 4.4 podemos ver que el terminal nos muestra que hemos entrado a éste y está a la espera hasta que recibe una instrucción. Si le enviamos un “1”, el modo automático comienza a funcionar. En el momento en el que el sistema recibe un “5”, el modo automático finaliza y se desconecta.

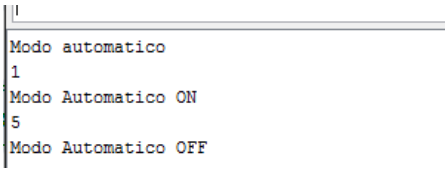

Figura 4.4: Comprobación del modo de funcionamiento automático.

Mediante estas comprobaciones se confirma que el vehículo puede acceder perfectamente a ambos modos de funcionamiento y realiza correctamente las instrucciones enviadas desde la App, quedando así comprobado el correcto funcionamiento de la recepción y tratamiento de datos.

Para comprobar ahora si la recepción y tratamiento de los datos por parte de la aplicación es correcta, en la figura 4.5 vemos que, en el movimiento manual, los resultados de las medidas se muestran perfectamente.

Figura 4.5: Comprobación modo manual en la App.

La comprobación del modo automático de la App se realiza de la misma manera que en el manual. Vamos a seguir el orden de acciones comentadas en el apartado 3.1 sobre el movimiento automático. Una vez conectados, pulsamos sobre el botón “play” y se reciben los primeros datos de temperatura y humedad medidos por el sensor y nos indica que se está comenzando el modo automático; de ahí pasamos a seguir líneas negras hasta encontrar una franja negra que nos hará cambiar de estado (figura 4.6).

Figura 4.6: Comprobación modo automático en la App (1).

Comenzamos el nuevo “entorno” configurando el sensor TCS230 y realizando una medida sobre blanco para calibrar el sensor (figura 4.7).

Figura 4.7: Comprobación modo automático en la App (2).

Tras esto, seguimos moviéndonos por líneas negras hasta encontrar otra franja negra que nos hará comprobar si está el “semáforo” delante o si se mide color del suelo. En la figura 4.8 se comprueba si hay algún obstáculo, como no lo hay, pasa a realizar una medida de color en el suelo.

La medición del color lo hacemos en RGB como se comentó en el apartado 2.3.2. La primera medida corresponde al rojo, ya que la cantidad de rojo es 190 y la de azul y verde rondan los 100.Y la segunda es verde porque hay 165 de verde, 135 de rojo y 113 de azul. Esta segunda

medida no es tan exacta ya que el sensor se ve afectado por la frecuencia infrarroja como ya se comentó en el apartado 2.3.2 de este trabajo.

Figura 4.8: Comprobación modo automático en la App (3).

Cuando se encuentra con el "semáforo" cambia de funcionamiento y espera a que éste se ponga verde para, finalmente, pasar al último entorno y comenzar a moverse autónomamente evitando obstáculos. En el momento en el que encuentra uno recalcula el camino para proseguir y en caso de no poder seguir por encontrarse totalmente cercado por obstáculos, finaliza el modo automático. Este funcionamiento lo vemos en la figura 4.9.

Figura 4.9: Comprobación modo automático en la App (4).

Ante los resultados obtenidos de las pruebas realizadas podemos concluir que el vehículo y la aplicación se comunican perfectamente y el funcionamiento del conjunto es el correcto.

Capítulo 5. Los costes de producción, el precio de venta y el umbral de rentabilidad.

5.1 Los costes de producción y el precio de venta al público.

Para la realización de este proyecto se ha tenido que incurrir, obviamente, en varios costes. Serán escasos, al ser el vehículo de pequeñas dimensiones, pero deben considerarse todos y tener la precaución de que no quede ninguno fuera, ya que en ese caso no se recuperaría, al no incorporarse al precio de venta, y podríamos llegar a tener pérdidas, porque los ingresos no recogerían todos los gastos o costes.

En los manuales de Economía se establece una primera y elemental clasificación de los costes empresariales, la que los divide en fijos y variables. Los fijos son aquéllos generados con independencia del volumen de producción: se produzcan más o menos vehículos, siempre estarán ahí, como el alquiler del local, algunos impuestos, la amortización de los ordenadores y demás maquinaria y elementos del inmovilizado, el sueldo del personal, fijo, estable, etc. Y los variables son los que varían en función del volumen de producción: si se producen más coches, habrá que comprar más placas, más ruedas, más leds, etc., e incluso contratar más personal.

Si mantenemos esta clasificación, los costes que supone la fabricación de este vehículo aproximadamente, ya separados en fijos y variables, serían los que aparecen en la siguiente tabla 5.1.

Concepto	**Precio (€)**
Fijos	
Coste por hora de la programación placa Arduino (50 horas dedicadas, a 70 €/hora)	3.500’00
Coste por hora de la programación en Android del dispositivo móvil (110 horas dedicadas, a 70 €/hora)	7.700’00
Consumo de energía eléctrica del PC (160 horas, a una media de 50 W/hora de consumo, a 0’22 €/Kwh (impuestos incluidos)	1’76
Amortización de la maquinaria (PC) (suponiendo un precio de compra del PC de 800 €, una duración de 3 años, 46 semanas de trabajo por año, 5 días semanales de trabajo, con 5 horas diarias de ordenador encendido, salen 138 semanas, 690 días, 3.450 horas, por lo que la *depreciación*, la pérdida, del ordenador será de 0’23 €/hora utilizado; al establecerse 160 horas de programación, supone una amortización de 36’80 €).	36’80
Amortización del resto del inmoviliario	1.000’00
Impuestos anuales y alquiler del local	9.000’00
Total costes fijos	**21.238’56**

Variables	
Chasis	13’00
Motores servo Futaba para las ruedas (2 motores, a 6 € c/u)	12’00
Motor servo SG90	6’00
Placa Board de 400 contactos (8’5 cm x 5 cm)	1’00
Pilas de 9V (3 pilas a 1 € c/u)	3’00
Soporte para Pilas de 9V (3 pilas a 0’7€ c/u)	2’10
Placa Arduino Mega	25’00
Cables macho-hembra y macho-macho (70 cables a 0’02 € c/u)	1’40
Leds (4 a 0’04 € c/u)	0’16
Zumbador	2’00
Módulo Bluetooth HC-05	3’50
Resistencias (6 a 0’08 € c/u)	0’48
PCB (placas de circuito impreso) (2 a 0’19 € c/u)	0’38
Sensor de ultrasonidos	1’90
Sensores TCRT5000 (2 a 0’126 € c/u)	0’25
Sensores TCS230 (2 a 3 € c/u)	6’00
Sensor de humedad y temperatura DHT11	0’86
Ruedas delanteras y la trasera	5’50
Piezas para sujeción y colocación del material	20’00
Coste por hora de instalación y montaje de todos los componentes (10 horas, a 30 €/h)	300’00
Total costes variables	**404’53**
Total costes	**21.643’09**

Tabla 5.1. Clasificación de los costes del vehículo.

Esto significa que si tuviéramos que realizar y vender al cabo del año un solo vehículo, siendo nosotros una empresa dada de alta en el censo de Hacienda y cumplidora de todas las obligaciones legales y formales, al menos tendríamos que gastarnos 21.643’09 euros para hacerlo. Bastante caro. Habría que estudiar el mercado y ver el precio de venta que tengan el resto de empresas competidoras nuestras sobre este mismo artículo, además de estudiar también la clientela para conocer si están dispuestos a pagar un precio de esa cuantía por el vehículo. Si cobramos por debajo de esos costes, lógicamente obtendremos pérdidas, que serían mayores cuanto más bajo fuera el precio de venta al público (PVP). Luego es necesario establecer ese PVP para conocer si nuestra empresa se podrá mantener en el mercado o desaparecerá en cuestión de meses; hay que conocer si nuestra empresa es “viable” económicamente o no.

No hemos incluido dentro de los costes variables unitarios el correspondiente al “semáforo”, porque es de suponer que la empresa que adquiera este vehículo para su utilización dentro de la misma, lo hará con independencia de la instalación física que requiera este producto, tanto de placas en el suelo con diferentes colores u otras variables para que el vehículo las siga, como de luces en paredes o postes para que el vehículo siga su trayectoria, instalación que iría a cargo de otra empresa. Por este motivo no se ha incluido el semáforo ni las cintas adhesivas como coste del producto.

Remitiéndonos de nuevo a los manuales de Economía más básicos, encontramos que el precio se puede establecer según diferentes criterios. Los más curiosos serían los siguientes:

- Según la “elasticidad” de nuestro producto. Un producto es elástico cuando al variar su precio en un porcentaje determinado su demanda varía en una mayor proporción, y es inelástico cuando varía en una menor proporción. El caso típico de uno elástico es el de los bienes normales o los de lujo, que si suben un poquito su precio, su demanda baja considerablemente, y si bajan de precio se venderían en mucha mayor medida. Los inelás-

ticos son los de primera necesidad, que por alto que esté su precio, su demanda seguirá siendo prácticamente la misma (caso de la gasolina, verduras, leche…).

- Observando el precio que tengan establecido las empresas de la competencia para ese mismo producto. En ese caso deberíamos situarlo a un nivel cercano a ese precio medio; si es superior no venderíamos, y si es inferior tal vez no llegáramos a cubrir los costes.
- Estudiando a nuestra clientela, sobre todo su renta, ya que si ésta es baja no podemos establecer unos altos precios para el artículo, y viceversa.
- En función de la oferta y la demanda: si detecto que el precio establecido es alto y los demandantes no llegan a comprar el artículo, no quedará más remedio que bajarlo; y si estaba bajo, podríamos intentar subirlo.
- Partiendo de los costes que tengamos para producirlo, con independencia de los criterios anteriores. Al coste total de producción le añadiríamos un porcentaje de beneficios y esa suma sería el PVP.

¿Qué criterio es el más correcto? Depende de cada empresa, de la situación del mercado, del tipo de producto que se intente vender, etc. En nuestro caso podría interpretarse que se trata de un producto de "lujo", de un bien que no es absolutamente necesario para sobrevivir, por lo que si el precio lo establecemos alto seguramente no tengamos grandes ventas, al menos en el primer año. Se trataría de un bien "elástico". Habría que realizar un estudio de mercado para comprobar cómo ven este producto los futuros o potenciales clientes y establecer un precio en consecuencia. Aunque este mismo producto tiene un destino distinto al particular: el empresarial, y para una empresa no se trata de un producto de lujo, sino quizás necesario para su mejor funcionamiento, por lo que sería un bien elástico y posiblemente estuvieran dispuestas a pagar un precio relativamente elevado por el vehículo, siempre que su rentabilidad fuera positiva.

La ley de la oferta y la demanda es otro criterio que hará que el precio se establezca en un nivel u otro; el vehículo tendrá que cambiar de precio según la información que reciba la empresa del mercado: si no se vende tendrá que bajarlo y si se observa que tiene una gran demanda podremos subir el precio.

Por último, si no atendemos a los criterios anteriores sino que nos basamos exclusivamente en nuestros costes, el PVP se establecerá en función de los mismos, añadiendo aquel porcentaje de beneficios sobre el total de costes unitarios, por unidad producida.

Todo lo anterior debe tenerse en cuenta a la hora de colocar el precio; no debemos basarnos en un solo criterio, aunque siempre habrá alguno que prevalezca ante los demás. En nuestro caso, al no conocer nuestro mercado ni la competencia, tendremos que partir de los costes y esperar a ver la reacción del mercado.

El resultado que obtengamos (beneficios o pérdidas) procede de la diferencia entre los ingresos y los gastos, de tal forma que

$$R = IT - CT \qquad \text{Ecuación 5.1}$$

$$IT = Q * P \qquad \text{Ecuación 5.2}$$

$$CT = CF + Cv*Q \qquad \text{Ecuación 5.3}$$

Donde R es el resultado, IT los ingresos totales, CT los costes totales, Q la cantidad vendida de vehículos, P el precio de venta, CF los costes fijos de fabricación del vehículo, y Cv los costes variables unitarios, por cada vehículo. No hay que confundir los costes variables unitarios (Cv), que son los costes que supone cada una de las unidades que se producen, con los costes variables totales (CV); éstos últimos son el resultado de multiplicar los variables unitarios por la cantidad de productos que se fabrican, de tal forma que CV= Cv*Q.

El PVP unitario, de cada vehículo, lo estableceremos partiendo de los costes variables unitarios (404'53 €) más una parte de los costes fijos, teniendo en cuenta que los posibles beneficios que pueda obtener el empresario ya se encuentran incluidos en el precio por hora que se

estableció para la programación, dentro de los costes fijos. De esta forma, si suponemos un porcentaje del 2'76% de los costes fijos (586 € aproximadamente, cantidad establecida al azar) para añadir a los variables unitarios, tendremos un PVP aproximado de 990 euros para nuestro vehículo. Si nuestro sentido común o cualquier indicio que nos muestre el mercado nos hacen suponer que este precio es demasiado alto o bajo, deberíamos modificarlo en ese sentido. Hemos de tener también en cuenta que es un precio "psicológico", no alcanza los 1.000 €, con lo que da la sensación de que es relativamente barato.

Una vez que tenemos ese precio con el que cubriremos todos los costes variables unitarios y una parte de los fijos, deberíamos preguntarnos cuántos vehículos hemos de vender para que no tengamos pérdidas, para que empecemos a tener beneficios, o sea, cuál es nuestro umbral de rentabilidad o punto muerto, ese punto en el que al vender una cantidad determinada de vehículos no tenemos pérdidas ni tampoco beneficios, nuestro resultado es cero; ése es el umbral de rentabilidad.

Si bajo estas condiciones vendiéramos sólo un vehículo al año, nuestros ingresos serían de 990 * 1 = 990 €, y nuestros costes de 21.238'56 + (404'53*1) = 21.643'09, por lo que el resultado sería una pérdida de 20.653'09 €. Si lográramos vender 5 unidades al año, el resultado sería de otra pérdida de 18.311'21, algo menor que la anterior. ¿Y cuál sería el resultado si vendiéramos 1.000 unidades? De un beneficio de 564.231'44 €. ¿Dónde está el punto de corte, el número de unidades en el que el resultado no sean ni pérdidas ni beneficios, el umbral de rentabilidad?

5.2 El umbral de rentabilidad o punto muerto.

Lo vamos a representar gráficamente; en el eje de abscisas se mostrará el volumen de producción o ventas, y en el de ordenadas los costes y los ingresos.

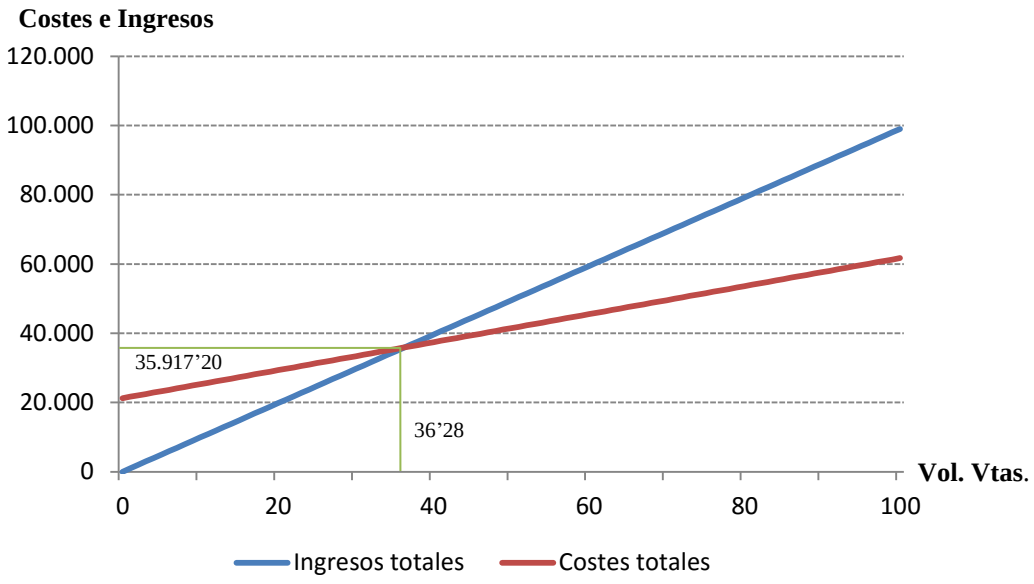

Gráfico 5.1: Umbral de rentabilidad.

La línea de los ingresos nace en el origen, puesto que si no vendemos nada, nuestros ingresos serán 0; si vendemos 10 unidades, serán de 9.900 €, si vendemos 67 unidades, los ingresos serán de 66.330 €, etc. Su pendiente dependerá del valor de P: si éste es alto, la pendiente será pronunciada, más vertical, y si el precio es bajo, la pendiente será más suave, más horizontal, y cortará a la curva de los costes totales más adelante, a un volumen de ventas superior, lo que querrá indicar que tenemos que vender más unidades para empezar a tener beneficios: eso no es

positivo, nos supondrá un mayor esfuerzo alcanzar el umbral de rentabilidad. Por este motivo, cuanto mayor sea la pendiente de los ingresos, cuanto mayor sea el precio de venta, antes conseguiremos el UR y antes empezaremos a tener beneficios. Eso es lo importante para la empresa, tener precios altos, pero al haber competencia no podremos conseguirlo fácilmente.

Por su parte, la línea de los costes totales nace en el eje de las ordenadas, puesto que aunque la venta sea de 0 unidades, la empresa siempre tiene unos costes fijos a los que hacer frente, y en nuestro caso son de 21,238'56 €. No tendremos costes variables unitarios puesto que no hemos producido ningún vehículo, pero sí contamos con aquellos fijos. Si vendemos 1 unidad, nuestros costes totales serán los fijos (21,238'56 €) más los variables de esa unidad (404'53 *1), total: 21,643'09 €; si vendemos 67 unidades, los costes totales serán de 48,342'07 €, etc. Esta recta nace desde los costes fijos, pero su pendiente depende de los costes variables unitarios, 404'53 € en nuestro caso. Si los costes fijos aumentan, la línea se desplazará hacia arriba manteniendo la misma pendiente, pero en este caso tendremos que vender más vehículos para alcanzar el UR, y si disminuyen será mejor para nosotros, puesto que lo alcanzaremos antes. Y si aumentan los costes variables, aumentará la pendiente, con lo que necesitaremos vender más productos para lograr el UR, y viceversa.

Por lo tanto, lo ideal para la empresa es alcanzar lo antes posible el UR, para de esta forma comenzar a obtener beneficios lo antes posible y reinvertirlos en nuevas materias primas. Que el UR sea más pequeño se logrará bien aumentando el precio de venta (subirá la pendiente de los ingresos) o bien reduciendo los costes fijos y/o los variables (disminuirá la pendiente de los costes totales, o la curva comenzará en una posición más baja del eje). Pero el precio de venta no se podrá aumentar fácilmente, y los costes tampoco se podrán reducir cómodamente. Luego será un problema para la empresa tomar una decisión en este sentido.

De todas formas, ¿cuál es el UR en nuestro caso? El punto en el que los ingresos son iguales a los costes totales, la posición en la que no existen pérdidas ni beneficios, es donde la curva de ingresos corta a la de los costes totales, o sea, donde se da la circunstancia de que

$$I_T = C_T \rightarrow Q * P = C_F + C_V * Q$$

Despejando de aquí Q se obtiene

$$Q = \frac{Costes\ Fijos}{P - Cv} = \frac{21{,}238'56}{990 - 404'53} = 36'28€$$

O sea, que hemos de producir y vender al año la cantidad de 36'28 vehículos para que empecemos a tener beneficios. Si vendemos menos, tendremos pérdidas, y cuantos más vendamos, mayores serán los beneficios.

Si subimos el precio a, por ejemplo, 1.100 euros, habría que vender sólo 30'54 vehículos al año para empezar a tener beneficios, lo mismo que si logramos reducir los costes variables al conseguir unos proveedores más baratos y de similar calidad, o los fijos, con menores alquileres, impuestos, nuestro sueldo, etc.

Si vendemos justos esos 36'28 vehículos al año, los ingresos y costes serían de 35.917'20 € (teniendo en cuenta todos los decimales), con lo que el resultado sería de 0, sin beneficios ni pérdidas, estaríamos situados en el UR. Pero si vendemos menos de esa cantidad obtendríamos pérdidas, y si por el contrario, vendiéramos más, obtendríamos beneficios.

Por todo ello, labor del ejecutivo de la empresa será el intentar sobrepasar ese umbral de rentabilidad aplicando las técnicas que la Economía de la Empresa nos puede enseñar, y hacer lo necesario para que sea lo más pequeño posible.

Capítulo 6. Aplicación del proyecto en el campo empresarial.

6.1 El momento de la *innovación.*

Es obvio que el actual resultado de este TFG tiene unas dimensiones físicas y unos costes tan reducidos que más se acerca a un entretenimiento juvenil que a un vehículo a utilizar en la industria; pero, gracias a esto, nos permite innovar y poder añadirle complementos para que realice otras tareas.

Su actual configuración es un paso todavía intermedio en la cadena de la innovación. Los capítulos 2 y 3 se dedicaron a estudiar qué materiales eran los más idóneos para su creación, qué sensores, motores y demás elementos habría que utilizar para cumplir los objetivos establecidos, cómo había que programar tanto el vehículo como el dispositivo para que pudieran conectarse, cómo se podría realizar precisamente esa conexión, etc. En fin, unos capítulos en los que, en principio, sólo se contemplaba la *investigación* para crear un vehículo que tuviera unas características determinadas, con independencia de que sus costes fueran astronómicos o reducidos, de que pudiera fabricarse en serie o no existiera tal posibilidad, de que tenga un público que quiera comprarlo o no, etc. Ya tenemos la primera letra (I, de Investigación).

Una vez que se han estudiado sus costes en el capítulo 5, se ha podido comprobar que perfectamente puede *desarrollarse* como un elemento industrial, que puede ser fabricado para su venta, que es viable económicamente y que puede ser introducido en el mercado como un producto novedoso que cumple con unos determinados servicios que el Departamento de marketing se encargaría de dar a conocer. Ya tenemos la segunda letra (D, de Desarrollo).

Con este capítulo 6 se pretende informar al "mercado" de la existencia de este producto e introducirlo en determinadas empresas. Se trataría de una *innovación* llevada a cabo por una empresa de un producto que presta un servicio que hasta ahora no ha sido muy generalizado, intentando captar parte del mercado que venía necesitándolo pero que no había suficientes empresas que lo ofrecieran. Ya tenemos la tercera letra (i, de innovación. Ya hemos realizado todo el proceso de la investigación, desarrollo e innovación, I+D+i.

6.2 Sectores industriales para su aplicación.

Este proyecto podría tener diferentes aplicaciones o trascendencia en el mundo empresarial. Su implementación en la industria tendría varios campos en donde fácilmente se asumiría por parte de las empresas. Por un lado, incrementaría la seguridad en los vehículos automóviles actuales; al incorporar ciertos sensores se controlarían con mayor precisión las situaciones de riesgo en la circulación, como las salidas de la calzada, posibles colisiones con otros vehículos, detección de peatones o animales, etc., al poder "obligar" al nuestro a frenarse en caso de acercamiento excesivo a esos objetos.

Por otro lado, como se encuentra conectado a una App nos mantendrá permanentemente informados sobre su estado, situación, etc., conociendo en cualquier momento cómo se encuentran elementos como la calidad de la luminosidad de las luces, los frenos, depósitos de líquidos, etc., por lo que sus posibles reparaciones necesitarían menor tiempo de detección.

Igualmente, al contar con sensores que reconocen ciertos colores y líneas en el suelo, se favorece su incorporación en robots o coches-robot de algunas industrias como puedan ser almacenes, hospitales públicos o privados, de transporte de material, etc., ya que según el color de las cintas en el suelo se podría dirigir a un lugar u otro. Por ejemplo, en el caso de un hospital, perfectamente podría dedicarse un vehículo de estas características a transportar informes a determinados Departamentos, llevar materiales a otros, suministrar los alimentos a los pacientes en las horas determinadas y en las habitaciones concretas, etc. En definitiva, la logística empresarial, paquetería, correspondencia, abastecimiento de repuestos y todo este tipo de actividades son las que en mayor medida se podrían beneficiar de los atributos de este vehículo.

La vigilancia tanto nocturna como diurna de viviendas particulares, organismos públicos, empresas, etc., sería otra posible aplicación de este vehículo si se le incorporara una cámara web y tuviera opción de conexión a Internet.

Tampoco habría que descartar su utilización por parte del ejército: con unos sensores determinados y específicos podría servir para detección de explosivos, análisis de cavidades, localización de ciertos materiales, etc.

Un caso más perfeccionado y actual es el correspondiente a la conducción del vehículo sin conductor de forma autónoma, como ya se ha implementado por parte de Google y Tesla en prototipos de vehículos autónomos. Otras empresas como Amazon y BMW utilizan vehículos para la logística industrial con un funcionamiento parecido al descrito en nuestro proyecto.

Capítulo 7. Conclusiones y líneas futuras de continuación del proyecto

A lo largo de los diferentes capítulos de este proyecto se ha diseñado e implementado un vehículo autónomo capaz de moverse en función del entorno y enviar y recibir información de una aplicación móvil.

Para la realización del mismo se han aplicado los conocimientos adquiridos sobre electrónica analógica, electrónica digital y programación adquiridos en la carrera a lo largo de los años. Por un lado, los correspondientes a la electrónica para el estudio, funcionamiento y ensamblaje de los componentes para realizar el vehículo y para la creación del circuito con las variables para que el vehículo las detecte. Y por otro, los conocimientos adquiridos en programación para realizar las correspondientes del vehículo y la aplicación. Estas programaciones fueron complejas, ya que requería de comunicación y tratamiento de los datos enviados entre ellas.

Finalmente la comunicación entre ambos elementos fue correcta y se consiguió el propósito del trabajo. Con todo esto se podría decir que el resultado obtenido en el TFG ha sido muy satisfactorio porque se ha conseguido desarrollar todo lo esperado.

Como posibles líneas futuras de continuación del proyecto se podrían destacar:

- Añadir más sensores para realizar mediciones de otras posibles variables a controlar, como un sensor Hall para medir campos magnéticos, o un sensor de agua para controlar la lluvia.
- Añadir más sensores de ultrasonidos para poder controlar más espacio. Esto nos podría facilitar una visión más amplia con lo que podríamos representar en la App un posible mapa.
- Cambiar el tipo de detección de color en el suelo. En nuestro proyecto se ha diseñado para que detecte un color y nos lo comunique, pero se podría detectar un camino mediante un color y dirigir el vehículo por él.
- Incorporar una cámara web para poder observar cualquier anomalía alrededor del vehículo, sirviendo de esta forma para vigilancia de locales, almacenes, etc.
- Cambiar el comportamiento del vehículo para que pueda pasar de un “entorno” a otro sin necesitar una configuración previamente realizada.
- Cubrir el vehículo con una carcasa para imprimirle una sensación de vehículo “más normal”.

Capítulo 8. Bibliografía

[1] Texas Advanced Optoelectronic Solutions (TAOS), "TCS230 Programmable color light-to-frequency converter". Febrero 2003.

[2] VISHAY, "TCRT5000" Datasheet. Agosto 2009.

[3] FUTABA, S3003 Servo Datasheet. Descargado en Marzo 2016.

[4] ITead Studio, "HC-05 Bluetooth Module" Datasheet. Junio 2010.

[5] D-Robotics UK, "DHT11 Humidity & Temperature Sensor". Julio 2010.

[6] ELECFreaks, "HC-SR04" Datasheet. Descargado en Marzo 2016.

[7] M. Banzi. Getting Started with Arduino. 2009

[8] Martínez Zaldívar, F. J.: apuntes de Android de la asignatura "Aplicaciones Telemáticas", Departamento de Comunicaciones, Facultad de Telecomunicaciones de la Universidad Politécnica de Valencia. Curso 2015-2016.

[9] Torres, Vicente: apuntes de Arduino de la asignatura "Aplicaciones de Microcontroladores", Departamento de Electrónica, Facultad de Telecomunicaciones de la Universidad Politécnica de Valencia. Curso 2014-2015.

[10] Alfaro Giménez, José, Clara González Fernández y Montserrat Pina Massachs (2016): *Economía de la empresa*, edit. McGraw-Hill, Madrid.

[11] Bueno Campos, Eduardo, Ignacio Cruz Roche y Juan José Durán Herrera (2007): *Economía de la empresa. Análisis de las decisiones empresariales*, edit. Pirámide, Madrid.

[12] Pérez Gorostegui, Eduardo (2002): *Economía de la empresa (introducción)*, edit. Centro de Estudios Ramón Areces, S.A., Madrid.

Capítulo 9. Anexos

Los códigos de programación correspondientes a este TFG se encuentran en los siguientes enlaces:

- Código del vehículo:

  https://www.dropbox.com/s/g5cct3fpmtkxnd7/codigo_vehiculo.rar?dl=0

- Código de la aplicación:

  https://www.dropbox.com/s/y1utcei3l6agfpb/codigo_app.rar?dl=0