BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA ELECTRÓNICA

“REPRODUCTOR DE FALLAS Y REGISTRO DE DATOS PARA AUTOMÓVILES DE LA MARCA VOLKSWAGEN”

TESIS

PRESENTADA PARA OBTENER EL TÍTULO DE

INGENIERO EN MECATRÓNICA

PRESENTA:

ISRAEL CISNEROS BONILLA

ASESOR:

M. C. DAVID CÉSAR MALPICA MOREDA

ABRIL DE 2016

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1

INTRODUCCIÓN

1.1 Identificación de la Problemática

En la gerencia de electrónica del área de desarrollo técnico de la empresa Volkswagen de México donde se realizan análisis de fallas en la electrónica del automóvil, por ejemplo en la parte de confort del usuario como son: luces, sistemas electrónicos del elevador de ventanas, quemacocos, retrovisores automáticos, etc., se realizan pruebas para verificar el correcto funcionamiento de estos sistemas.

Sin embargo existen ocasiones donde las fallas son muy difíciles de reproducir, debido a que no se cuenta con los datos de las condiciones exactas en que aparecieron dichas fallas, ya que éstas son esporádicas y se tiene que repetir el procedimiento un amplio número de veces por una persona, usualmente un especialista de la empresa que está dedicado al análisis del problema intentando volver a reproducir la falla por un largo periodo de tiempo. Esta actividad consume tiempo del especialista y dinero a la gerencia.

1.2 Objetivo general

Construir un dispositivo que reproduzca un caso de prueba específico, generando y grabando señales eléctricas y señales CAN para ser implementado en cualquier plataforma de los automóviles de la familia Volkswagen.

1.3 Objetivos específicos

- Comunicar el automóvil con un microcontrolador a través del BUS CAN y registrar los datos de los mensajes CAN en una memoria externa, con una impresión de tiempo de lectura.
- Acondicionar las señales eléctricas multipropósito del dispositivo y mostrar los valores de algunas señales en un LCD, además de registrar la información de todas las señales en una memoria externa.
- Aplicar el dispositivo para un caso real en la empresa Volkswagen dentro del área de desarrollo técnico.
- Elaborar el manual de usuario del dispositivo y la programación para un caso de prueba.

1.4 Justificación

Cuando un cliente o un especialista encuentran una falla en los dispositivos electrónicos del automóvil se realizan pruebas con base a un catálogo donde se indica qué acciones realizar para hallar la falla, sin embargo estos pueden ser hasta cientos de casos y en ocasiones difíciles de reproducir para el especialista. Con éste dispositivo se encontrará en menor tiempo las condiciones que provocan una falla y se tendrá registro preciso de cómo ocurrió la anomalía en el automóvil, grabando las señales CAN, digitales y analógicas (Estas señales podrán ser utilizadas en otro programa propiedad de la empresa para su análisis). Por su propio diseño, podrá reproducir condiciones muy difíciles de realizar por una persona como generar señales en ciertas ventanas de tiempo.

Todo lo anterior reducirá el costo del análisis de fallas y el tiempo que le toma al especialista reproducirlas, ya que actualmente le lleva hasta una semana de trabajo volver a encontrar las condiciones en las que se produce la falla. Además el sistema a desarrollar será amigable y de fácil uso para el especialista, así como portátil.

1.5 Descripción del proyecto

El presente trabajo tiene como finalidad el desarrollo de un dispositivo que pueda leer 2 señales CAN de clase C (High Speed) provenientes de un automóvil de la plataforma MQB o PQ35 que son las utilizadas actualmente en la producción de automóviles VW, por lo cual la velocidad de comunicación será configurable, además que deberá contar con entradas multipropósito analógicas y digitales que sirvan para accionar sistemas o simular ciertos dispositivos del automóvil.

Este sistema integrará el acondicionamiento de las señales eléctricas hacia los dispositivos que vayan a activar o a leer del automóvil, ya que los voltajes utilizados en la electrónica automotriz comúnmente son de 12 V. Las salidas y entradas analógicas tendrán una resolución de 8 bits, y además es posible el aumento del número de salidas del dispositivo con multiplexores si llegara a requerirse. El sistema incorporará un LCD que mostrará mensajes de utilidad al especialista para su prueba. Finalmente toda la información será guardada en una memoria externa con una impresión de tiempo del instante en el cual fue tomada para su posterior análisis.

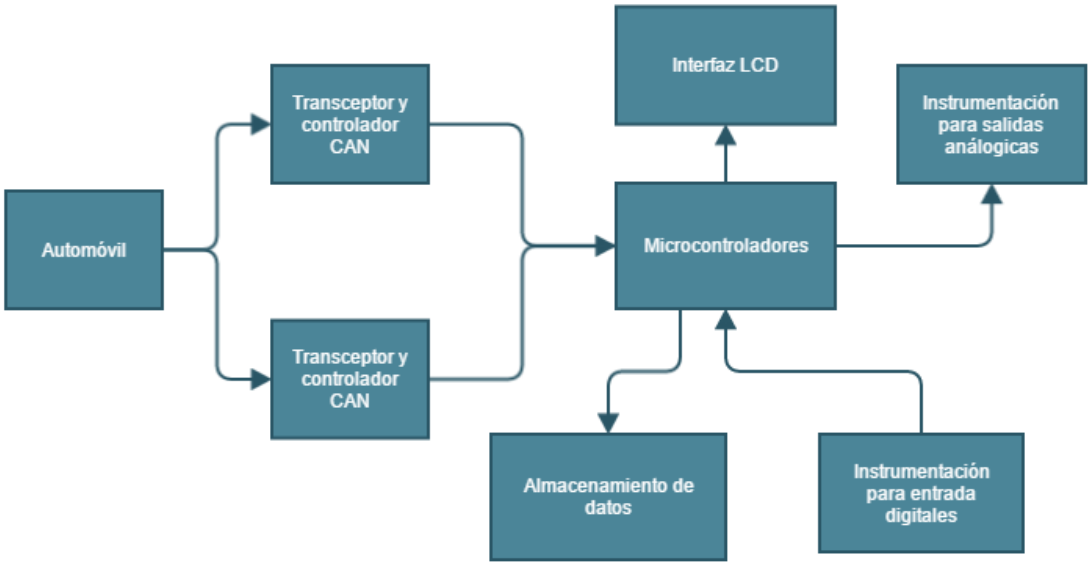


Figura 1.1 Diagrama a bloques del dispositivo.

En la figura 1.1 se muestra un diagrama general de lo que será el dispositivo, en donde se puede observar los elementos mencionados anteriormente. El sistema se conforma por un par de transceptores y controladores que obtendrán los mensajes de la red CAN; cada par de estos integrados tendrán un microcontrolador asociado con el que se comunicarán para enviar los mensajes y posteriormente ser mandados a un módulo con una memoria SD en el que se generarán archivos de texto con la información de los mensajes. También se muestra el bloque de la interfaz LCD como se mencionó anteriormente en la que se visualizará información de inicialización o valores de entradas multipropósito y finalmente los bloques de la instrumentación de las salidas multipropósito digitales y analógicas.

Se realizará la programación de un caso de prueba donde abarque la mayor parte de estas entradas y salidas digitales. El sistema no contará con una interfaz gráfica de programación, la programación se realizará directamente en un compilador por lo que tendrá que ser amigable con el usuario para que éste reprograme de una manera fácil las funciones de las entradas multipropósito.

CAPÍTULO 2

COMUNICACIÓN AUTOMÓVIL MICROCONTROLADOR CON LA RED CAN

2.1 Introducción

La mayoría de las unidades de control de los sistemas modernos del automóvil que busca confort, seguridad, tracción del motor, reducción de combustible y emisiones contaminantes requieren de intercambio de una gran información a través de la red CAN.

En este capítulo se describe brevemente la red CAN con la finalidad de entender el protocolo de comunicación y posteriormente se implementa en un microcontrolador ATmega de la placa Arduino con ayuda de una placa externa que contiene un controlador y transceptor.

Se le imprime el tiempo de lectura del mensaje y se guardan los mensajes en una memoria SD, convirtiendo a la placa Arduino en un datalogger de bajo costo.
El dispositivo se conecta con un conector DB9 a la red CAN física del automóvil.

2.2 Red CAN

En la actualidad existen varias decenas de protocolos automotrices. Entre los más destacado se encuentra el protocolo CAN; éste se distingue por ser de baja velocidad, de bajo costo e interconecta a las unidades de control de una forma global aumentando la información que viaja en esta red.

Por ejemplo el área de la tracción del automóvil forma un sistema global y se conforma por:

1. La unidad de control del motor.
2. La unidad de control para cambio automático.
3. La unidad de control ABS.

En el área de confort del automóvil constituye también un sistema global y se conforma por:

1. La unidad de control central.
2. Las unidades de control de puertas.

En la figura 2.1 se muestran algunas de las unidades de control del automóvil [1].

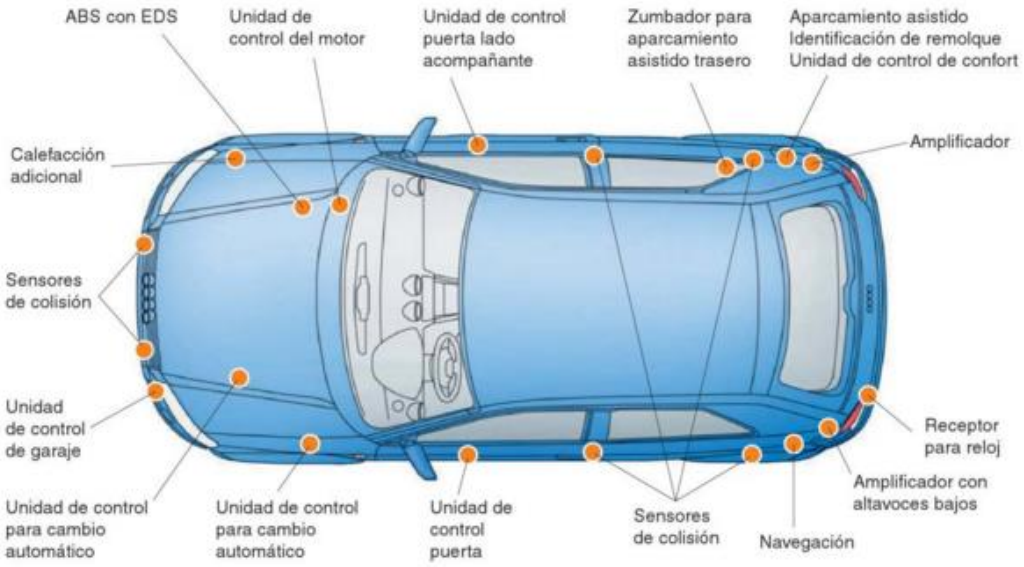


Figura 2.1 Ejemplo de disposición de las unidades de control en Audi A3.

Las ventajas del BUS de datos se enlistan a continuación:

1. Si se amplía el protocolo solamente se necesitan modificaciones en el software.
2. Un bajo porcentaje de errores mediante una verificación continua de la información transmitida, de parte de las unidades de control, y mediante protecciones adicionales.
3. Menos sensores y cables de señales gracias al uso múltiple de una misma señal.

4. No afectan las alteraciones magnéticas y otros tipos de interferencia a la red de comunicación.
5. Es posible una transmisión de datos muy rápida entre las unidades de control.
6. Más espacio disponible, mediante unidades de control más pequeñas y conectores más compactos.
7. El BUS CAN de datos está normalizado a nivel mundial.

2.3 Descripción del protocolo de comunicación CAN (Controller Area Network)

Es una red serial que originalmente fue diseñada para la industria automotriz, sin embargo se ha popularizado en aplicaciones industriales, sistemas embebidos, y sistemas donde la comunicación es en tiempo real.

El BUS CAN emplea dos líneas eléctricas para representar los bits dominantes y los bits recesivos, estos son nombrados CAN_Low y CAN_High y normalmente se emplea cable UTP (*Unshielded Twister Pair*). La velocidad de transferencia alcanza hasta los 100 Mb/s [2].

Figura 2.2 Cable de red CAN.

Las tramas están conformadas por 4 tipos de mensajes que pueden ser enviados a través del CAN.

1. La trama de datos
    - Campo de inicio SOF: Contiene un bit, su función es indicar cuando comienza la comunicación y sincronizar los nodos.
    - Campo de arbitraje: Cada nodo cuenta con un identificador único, decide qué mensaje se transmite primero. Se conforma por 11 bits.
    - Campo de control: Se conforma por los bits de petición remota RTR, el bit de identificación IDE y un bit reservado.

- Campo de tamaño de mensaje DLC: Especifica el número de bytes en el mensaje y se conforma por 4 bits
- Campo de datos: Es la información de nuestro mensaje y se conforma por 8 bytes.
- Campo CRC , código de redundancia cíclica (15 bits)
- Campo de confirmación ACK: Informa si el mensaje ha sido recibido exitosamente.
- Campo de fin de programa OEF: Indica el término de la transmisión y se conforma por 7 bits
- Campo de espacio entre tramas: Son 2 bits que consumen tiempo en la transmisión para que el mensaje sea almacenado por el receptor.

En la figura 2.3 se muestra el datagrama [3]

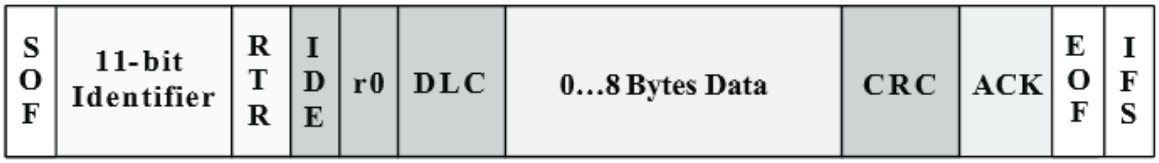


Figura 2.3 CAN estándar.

2. La trama remota: Solicita información a otro nodo en la red, es muy similar a la trama de datos.
3. Trama de error: Es trasmitido cuando un nodo detecta errores en el mensaje, causando que los demás nodos envíen una trama de error.
4. Trama de sobre carga: Su función principal es causar un retraso entre tramas, para que se realice el procesamiento correctamente.

2.4 El BUS CAN

La conexión física entre un microcontrolador o un DSP para acceder al BUS es a través de un transceptor y un controlador.

El transceptor adapta los niveles lógicos TTL a señales con niveles utilizados en el BUS CAN. La figura 2.4 muestra este cambio [2].

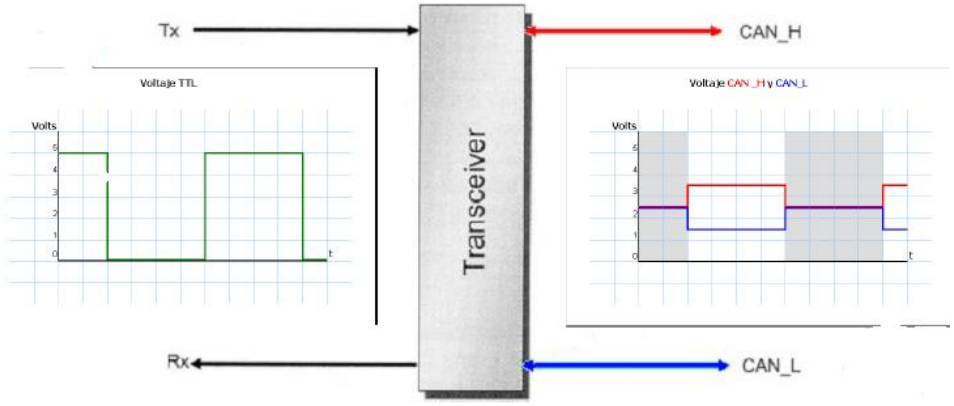


Figura 2.4 Adaptación de niveles de voltaje.

El controlador traduce y manipula los mensajes recibidos y enviados en el BUS además de que gestiona las señales en el mismo.

En la figura 2.5 se muestra un diagrama general de la conexión entre los nodos [3].

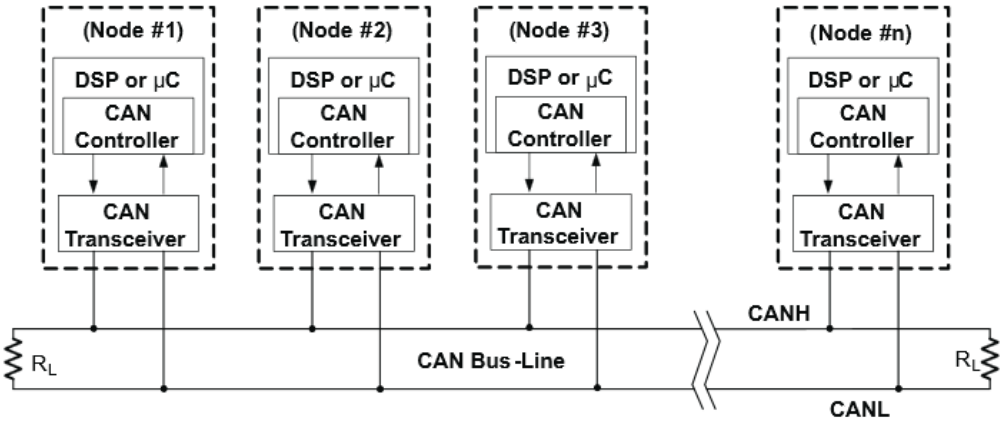


Figura 2.5 Adaptación de niveles de voltaje.

2.5 CAN-BUS Shield

Los Shield son tarjetas que le brindan características adicionales a las tarjetas Arduino. El CAN-BUS Shield le brinda la capacidad al Arduino de comunicarse con un automóvil y obtener información de las unidades de control del automóvil; también se puede enviar datos a las unidades de control.

Este Shield implementa el controlador MCP2515 con el transceptor MCP2551 de la marca Microchip, el cual cuenta con un conector DB9 que puede ser utilizado con una interfaz DB9 a OBD-II.

Entre las características importantes:

- Protocolo CAN V2.0 de velocidad configurable de hasta 1 Mb/s.
- Interfaz SPI de alta velocidad (10 MHz).
- Entrada para memoria SD.
- Conexión para LCD serial.

En la figura 2.6 se muestra el Shield CAN. El diagrama del circuito del Shield se encuentra en el primer anexo.

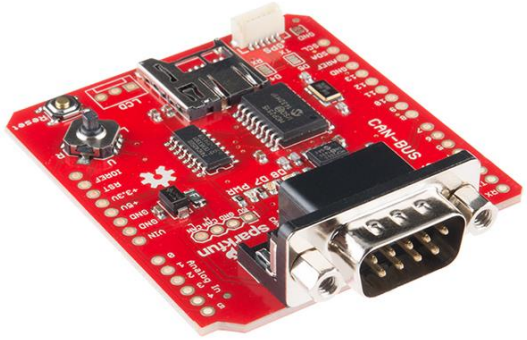


Figura 2.6 BUS CAN Shield de la marca SparkFun.

En la figura 2.7 se muestra el diagrama a bloques del circuito utilizando un Arduino Mega y un Arduino UNO con el BUS CAN Shield.

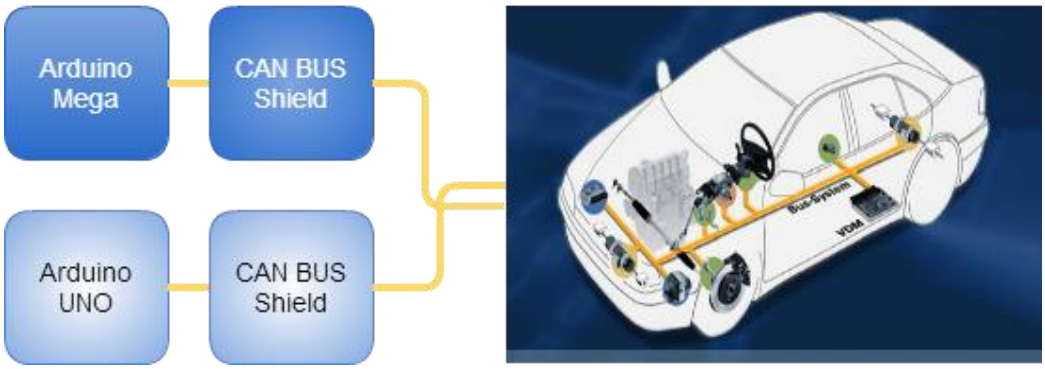


Figura 2.7 Diagrama a bloque de la conexión del BUS CAN

2.6 Comunicación entre Arduino y automóvil por el BUS CAN

2.6.1 Configuración de IDE de Arduino

Se utilizara el IDE de Arduino para realizar la programación del dispositivo. Para facilitar su programación se utilizarán librerías que ofrece el fabricante SparkFun para poder utilizar el Shield con el Arduino; la primera librería se descargará de la página oficial de SparkFun, que se puede encontrar en el siguiente link: https://learn.sparkfun.com/tutorials/can-bus-shield. La función es configurar el controlador MCP2515 y el transceptor MCP2551.

El procedimiento para instalar la librería en el IDE de Arduino es el siguiente:

1.- En el IDE de Arduino, en el menú programa, seleccionar la opción incluir librería; dentro de ésta opción, seleccionar agregar librería .ZIP, en la ventana que aparece se puede elegir la librería, agregándose al final de todas las librerías ya existentes en el IDE. En la figura 2.8 se muestra la ruta a seguir en el IDE.

Como se mencionó anteriormente el controlador cuenta con una interfaz SPI para comunicarse con el Arduino [4]. En el IDE se utilizará la librería con el nombre SPI (Serial Peripheral Interface) que ya se encuentra instalada y que es

necesaria para poder utilizar el Shield. El usuario solamente debe seleccionarla de la lista de librerías instaladas en el Arduino IDE.

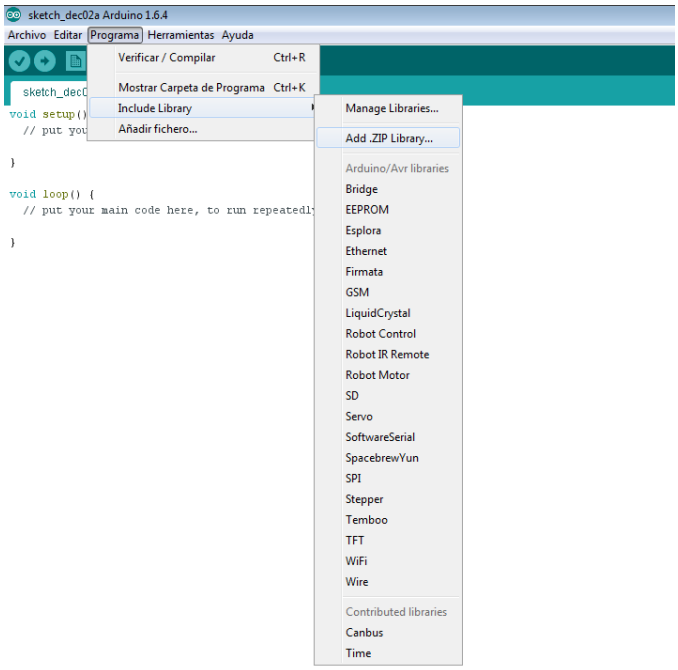


Figura 2.8 Ruta de instalación de librerías en el IDE de Arduino.

2.6.2 Funciones de la librería CANBUS

La librería que ofrece el fabricante cuenta con funciones que podremos implementar directamente en el programa para poder leer los mensajes en el BUS CAN.

1. **Canbus.init:** Inicializa la velocidad del controlador MCP2515 a una velocidad especificada. En la siguiente lista se muestran las velocidades a las que se puede configurar.
    - CAN_100 kb/s
    - CAN_125 kb/s
    - CAN_250 kb/s
    - CAN_500 kb/s

Inicialmente en la librería del fabricante tiene disponibles 125 kb/s, 250 kb/s y 500 kb/s, por lo que se tendrá que modificar el archivo CANBUS.H que se encuentra dentro de la librería que se instaló anteriormente y agregar la velocidad de 100 kb/s, agregando la siguiente línea “#define CANSPEED_100 9”, en la parte de las directivas.

2. **Check_message:** Indica si se ha recibido algún nuevo mensaje desde el bus en el controlador CAN, que no ha sido leído aún. Devuelve un valor booleano.
   0: Si no existe ningún mensaje.
   1: Si existe algún mensaje.
3. **Get_message:** Devuelve el número de mensajes disponibles en el buffer de recepción. Los mensajes recibidos esperan a ser leídos y almacenados en el buffer de entrada.
4. **Message.id**: Devuelve el identificador del mensaje CAN.
5. **Message.header.length**: Devuelve el tamaño del mensaje CAN leído.
6. **Message.data**: Devuelve los bytes del dato según sea el tamaño del mensaje CAN.

Las funciones 4, 5, 6 son acompañadas por un Serial.print para ser visualizadas en el monitor serial del IDE de Arduino. El código de la implementación de estas funciones para la lectura de los mensajes CAN se encuentra en el anexo 2, este código permite leer mensajes de la red CAN.

2.6.3 Almacenamiento de Mensajes CAN en memoria microSD

El Shield de BUS CAN incorpora una ranura de memoria microSD, y se implementa con una librería que se encuentra ya instalada en el IDE de Arduino con el nombre SD, ésta librería nos permite leer y escribir datos en memorias SD. La librería soporta sistema de archivos FAT16 y FAT32.

Los nombres que se pueden utilizar con ésta librería utilizan el formato 8.3, con el cuál el nombre debe ser 8 caracteres y extensiones de tres, no debe llevar acentos o espacios en blanco.

La memoria antes de ser utilizada debe ser formateada con el sistema de archivos FAT 32, la ruta se muestra en la figura 2.9.

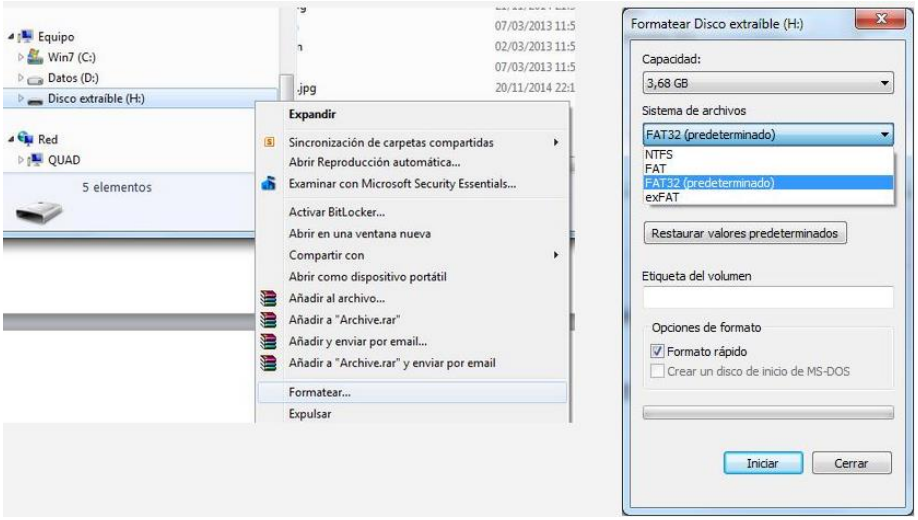


Figura 2.9 Ruta para formatear memoria FAT32.

Las funciones que ofrece la librería son las siguientes:

1.- SD.begin(): Inicializa la librería SD, la memoria SD y el BUS SPI. Las terminales del BUS SPI en el Arduino UNO se encuentran en los pines digitales 11, 12, 13, la selección del esclavo es en el pin 10, definido por el fabricante. Para el Shield de CAN, el pin para iniciar comunicación con la memoria SD es el número 9. En el Arduino MEGA las terminales del BUS se encuentran en los pines 50, 51 y 52, la selección se hace por medio del pin 53; para la selección de la memoria SD es el pin 9. Los pines se definen en la tabla 2.1.

Nombre	Pines Arduino MEGA	Pines Arduino UNO
CS	53	10
SCK	52	13
MOSI	51	11
MISO	50	12
INT	19	2

Tabla 2.1 Pines en Arduino UNO y MEGA del BUS SPI.

2.- Open(): Abre un archivo en la memoria SD si el archivo existe, si el archivo no existe esta función creará un archivo con el nombre que sea especificado, si no se crea nada devolverá un valor booleano 0. La extensión que puede tener el archivo es TXT o XLS.

Para crear el archivo en la memoria se utilizará un tipo de dato archivo que utiliza la biblioteca de la clase **"**File", éste será definido al inicio del código. Este tipo permite leer datos y escribir en la memoria SD.

3.- Print(): Permite imprimir los datos en la memoria SD. Para poder utilizar está función es necesario ya tener abierto el archivo en la SD.

Ejemplo: *file.print(datos*), como se observa file se refiere al archivo en el cual se van a grabar los datos.

4.- Println(): Tiene la misma función que print, sin embargo terminando de grabar los datos, hace un retorno de carro, para comenzar a grabar datos en una nueva línea.

5.- Close(): Cierra el archivo en el cual se están grabando los datos, se debe asegurar que el archivo es cerrado de lo contrario no grabará ningún dato.

Con estas funciones se realizará el grabado de los datos de la red CAN. En el código del anexo 2 han sido agregadas estás funciones. El nuevo código se encuentra en el anexo 3.

2.6.4 Impresión de tiempo a mensajes CAN y biblioteca TIME

Los mensajes CAN necesitan tener una impresión del tiempo del cual fue leído y que sea grabado en la SD. Se utilizará la función millis() la cual retorna el número de milisegundos a partir de que el Arduino fue encendido; el timer al cual está relacionado la función se desborda en aproximadamente 50 días. El valor de esta función será asignado a una variable la cual será manipulada para imprimir el tiempo en segundos a cada mensaje de CAN. Con ayuda de la función print se grabará en la SD.

Es necesario que el archivo de la lectura de mensajes CAN tenga una impresión de la fecha en la cual se está realizando. Se utilizará la biblioteca TIME que será necesario instalar en IDE de Arduino.

Esta biblioteca utiliza el tiempo interno de Arduino y devuelve desde segundos hasta meses, sin embargo utiliza el timer que está relacionado con la función millis por lo cual se desbordará después de aproximadamente 50 días y la fecha se reiniciará.

La función que se declara es *time_t T = now()* que devuelve el valor interno del Arduino, posteriormente se utiliza la función *SetTime* con la cual se podrá definir la fecha desde la cual empezará el Arduino a contar el tiempo transcurrido.

Para grabar la fecha en la memoria SD se usará la función print y se agregará el formato que se desee, por ejemplo *File.print(day(t))*; como se puede observar es una función dependiente del tiempo. El código se muestra en el anexo 4.

2.6.5 Modificación de librería CAN

Es necesario utilizar 2 Arduinos para leer 2 canales simultáneamente. Como se explicó anteriormente el controlador CAN se comunica con el Arduino por medio del protocolo SPI, a pesar de que se puede hacer una red maestro esclavos y ofrece una velocidad alta de transferencia, cada esclavo se comunicará a la vez con el maestro, es decir no puede existir comunicación simultánea.

Se utiliza un Arduino UNO para únicamente leer mensajes y guárdalos en la memoria SD y un Arduino MEGA que realizará la misma función que Arduino UNO sin embargo se encargará de la parte de las salidas multipropósito que se explicarán en el capítulo 4. Se conectará el Arduino UNO al Arduino Mega para saber cuando ya ha comenzado a recibir mensajes o los está grabando en la memoria SD provenientes de su BUS de CAN.

La librería de BUS CAN está diseñada para el Arduino UNO, sin embargo para el Arduino MEGA es necesario modificar los valores de unos pines de los puertos que utiliza la biblioteca, ya que cuando el programa trate de acceder a ellos ésta no funcionará y no podrá conectarse el controlador con el

microcontrolador del Arduino. Este archivo define los pines de puerto SPI y pines para el controlador MCP2515.

El archivo se encontrará en una ruta que el IDE de Arduino genera automáticamente cuando se instala en cualquier computadora con Windows. La ruta es la siguiente: D:\Mis, Documentos\Arduino\libraries\Canbus; dentro de esta carpeta se encuentra el archivo defaults.h; antes de modificar el archivo es necesario saber el nombre del puerto y número de pin asociado al Arduino.

En la figura 2.9 se muestra un esquema del Arduino UNO y en la figura 2.10 el esquema de los conectores inferiores del Arduino MEGA ya que en estos pines se encuentran las terminales del puerto SPI que se tendrá que cablear y declarar. Los cuadros amarillos contienen el nombre del puerto y número de pin, en el archivo defaults se renombrarán los pines en sus respectivas directivas y se cargará el código para el Arduino MEGA.

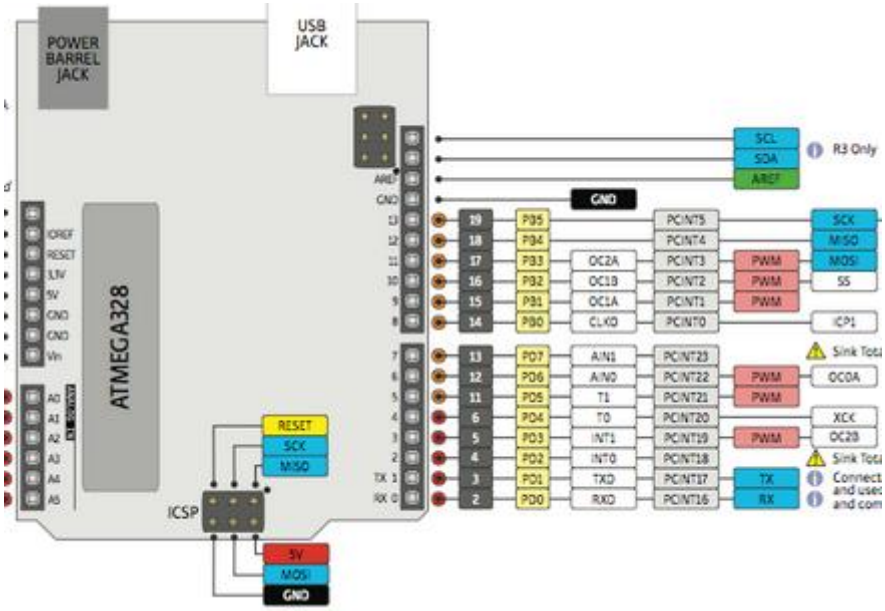


Figura 2.10 Esquema de Arduino UNO.

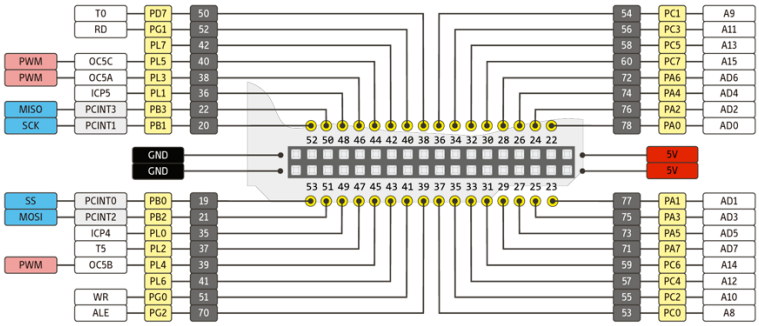


Figura 2.11 Conectores inferiores de Arduino MEGA.

Finalmente en la figura 2.11, del lado izquierdo, se muestra el archivo modificado y del lado derecho el archivo original.

```
#ifndef DEFAULTS_H
#define DEFAULTS_H
 //LIbreria Arduino MEGA
#define P_MOSI   B,2
#define P_MISO   B,3
#define P_SCK    B,1
//#define MCP2515_CS      D,3 // Rev A
#define MCP2515_CS       B,0 // Rev B
#define MCP2515_INT      E,4
#define LED2_HIGH        H,4
#define LED2_LOW         H,4

#endif  // DEFAULTS_H
```

```
#ifndef DEFAULTS_H
#define DEFAULTS_H
// Libreria Arduino UNO
#define P_MOSI   B,3
#define P_MISO   B,4
#define P_SCK    B,5
//#define MCP2515_CS    D,3 // Rev A
#define MCP2515_CS       B,2 // Rev B
#define MCP2515_INT      D,2
#define LED2_HIGH        B,0
#define LED2_LOW         B,0

#endif  // DEFAULTS_H
```

Figura 2.12 Archivo modificado de la librería CAN BUS.

2.7 Resultados

En la figura 2.12 se muestra el esquema de conexión para el Arduino uno, la ventaja que presentan los Shields es que la mayor parte de ellos están diseñados para el Arduino UNO y solamente se debe insertar en los pines correspondientes y la figura 2.13 muestra la conexión para el Arduino MEGA con el BUS CAN.

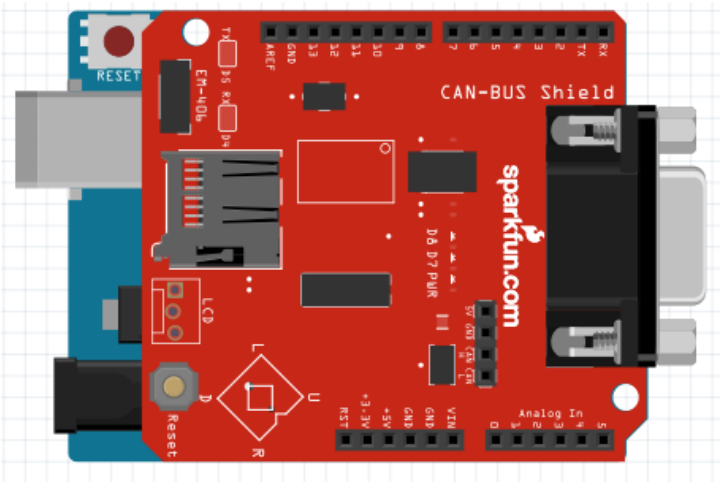


Figura 2.13 Arduino UNO con BUS CAN Shield.

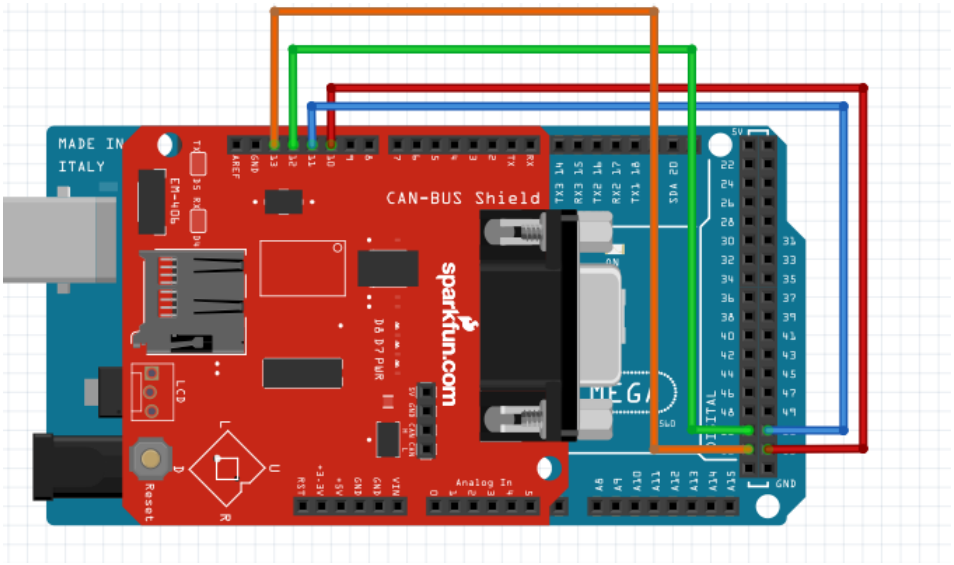


Figura 2.14 Arduino MEGA con BUS CAN Shield.

En la figura 2.14 se muestra el monitor serial del IDE de Arduino, donde se imprimen los mensajes de inicialización de SD y de BUS CAN. Este mensaje se refiere a que el Arduino ya se conectó con el transceptor y está listo para leer los mensajes del BUS CAN; si el archivo defaults.h no fue modificado para el Arduino

MEGA, esté no podrá iniciar la comunicación con el controlador CAN. Los mensajes son grabados en la memoria SD como un archivo de extensión TXT.

La velocidad del BUS CAN confort es a 500 kb/s para un automóvil GOLF A7

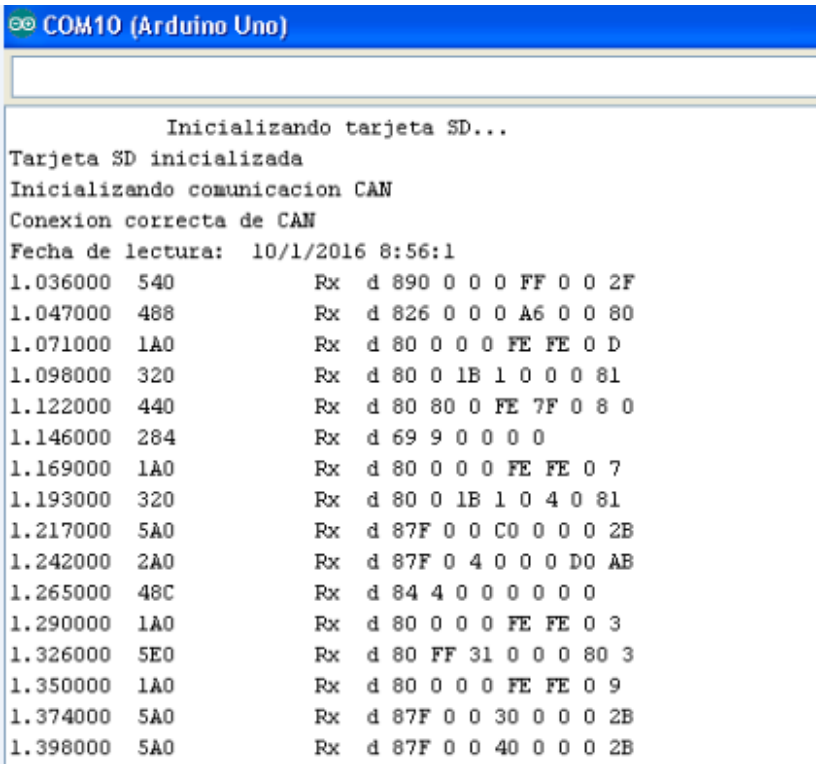


Figura 2.15 Monitor Serial de Arduino IDE con datos de lectura.

2.8 Conclusión

Se comunicó un automóvil y un microcontrolador ATmega de una tarjeta Arduino, además se registraron los datos de los mensajes CAN en una memoria externa con una impresión de tiempo de lectura, los mensajes se visualizan en el monitor serial del IDE de Arduino o en el archivo que se genera en la memoria.

La velocidad de transmisión de mensajes es configurable con los valores de 100 kb/s y 500 kb/s utilizados comúnmente en las redes automotrices CAN.

La velocidad mínima promedio de muestreo es de 25 ms, siendo una velocidad promedio aceptable para análisis de fallas.

CAPÍTULO 3

INTERFAZ LCD

3.1 Introducción

Cuando no se tiene una computadora como intermediario entre el Arduino y las variables externas es necesario recurrir a una interfaz sencilla y de fácil uso. El LCD (Liquid Cristal Display) ofrece una manera rápida de mostrar mensajes al usuario, en el mercado se encuentran dos tipos de LCD que muestran únicamente caracteres basados en el código ASCII o gráficas con la capacidad de mostrar imágenes y caracteres.

En el presente capítulo se muestra la implementación de una pantalla LCD con interfaz serial con un Arduino y la implementación de funciones basadas en el protocolo UART.

3.2 LCD serial

Las pantallas LCD comunes tienen el inconveniente de requerir muchos cables para conectarse al circuito. Como consecuencia, en la placa Arduino pueden quedar pocos pines disponibles para ser usados. Una solución a este inconveniente es el empleo de pantallas LCD que utilizan un sistema de comunicación con la placa Arduino, como pueden ser los protocolos I2C (el cual solo utiliza las líneas SDA y SCL), o el serie (usando solo las líneas RX y TX), principalmente.

Las pantallas LCD que implementan el protocolo serie para comunicarse con el Arduino tienen incorporado un chip TTL-UART, de esta manera solamente

necesitan 3 cables: el de alimentación a la pantalla, tierra y la conexión del pin RX de la pantalla al pin TX de puerto serial de la placa Arduino.

Los comandos son enviados con ayuda de la función Serial.write para realizar acciones en la pantalla como variación de la iluminación de pantalla, limpiar pantalla o posicionar el cursor etc. Serial.print se utilizará para imprimir texto. Estas funciones pueden variar dependiendo el fabricante.

3.3 Conexión del módulo LCD Protheo

El módulo de desarrollo LCD Protheo del fabricante “Cosas de Ingeniería” es una solución para el fácil manejo de cualquier LCD 16x2 o 20x4. Únicamente se debe insertar el LCD en los pines correspondientes y alimentar con un voltaje de 4 V a 6 V en el pin indicado y la tierra en el pin de GND, el módulo configurará la LCD para el control del dispositivo a través de comandos seriales.

El dispositivo no cuenta con una protección contra corto circuito por lo tanto se debe tener especial cuidado en conectar adecuadamente la alimentación ya que una mala conexión puede dañar el dispositivo.

Para su control se utiliza el pin Rx, el cual se encargará de recibir los datos seriales e interpretarlos para mostrarlos en la pantalla. La comunicación serial se realiza a 9600 baudios; el módulo ofrece un pin TX por si se requiere recibir retroalimentación del LCD con los datos enviados.

3.4 Inicialización el modulo LCD Protheo

Una vez que el módulo esté adecuadamente conectado se procederá a energizar el circuito. Cuando el módulo arranque empezará a mostrarse la introducción en el LCD del fabricante, lo que indica que se han realizados adecuadamente las conexiones al dispositivo. Al mismo tiempo el módulo enviará, a través de su pin de Tx, el dato (@) que indica que la inicialización del LCD está lista para comenzar a recibir datos.

Cuando la introducción termina su secuencia, el LCD se limpiará y aparecerá en la primera posición el cursor parpadeando esperando a recibir datos. A partir de este momento se podrán enviar datos en código ASCII que serán interpretados por el LCD y mostrados en pantalla, escribiéndolos de izquierda a

derecha. Una vez que haya llenado las primeras 16 posiciones, continuará escribiendo en la memoria del LCD en un display virtual hasta completar 40 caracteres, esto para el LCD 16x2; para el LCD 20x4 se mostrarán 20 caracteres.

Estos últimos caracteres estarán presentes en el LCD pero no se mostrarán a la vista. Se podrán mostrar estos caracteres utilizando los comandos de rotar a la derecha o rotar a la izquierda.

3.5 Comandos utilizando el protocolo UART

El módulo acepta comandos mediante el protocolo UART (Universal Asynchronous Receiver Transmitter) con los parámetros siguientes: Velocidad a 9600 baudios por segundo, 8 bits, sin paridad. Todos los comandos deben finalizar con un retorno de carro (‘0x0D’).

A continuación se muestran en la tabla 3.1 algunos de los comandos útiles con sus respectivas funciones.

Función	Comando	Descripción
Inicialización	@I20X4	Inicialización de LCD 20x4
Borrar LCD	@B	Borra el contenido del LCD y manda el cursor a la posición 1,1.
Posición	@Py,x	Coloca el cursor en la posición Y, X. Donde, X e Y son números de los renglones en código ASCII
Imprimir un dato.	@Dx	Imprime un dato en el LCD (1 Byte).
Cadena de datos.	@E	Imprimir una cadena en el LCD.
Corrimiento izquierda	@CDI	Desplaza el contenido a la izquierda.
Corrimiento derecha	@CDD	Desplaza el contenido a la derecha.

Tabla 3.1 Comandos utilizando el protocolo UART.

Estos comandos están limitados a funciones básicas del LCD, sin embargo son muy útiles en proyectos donde es necesario mostrar mensajes de una manera sencilla con un LCD tipo serial.

3.6 Programación de Arduino con LCD serial.

La función que realizará el LCD es mostrar mensajes de inicialización de comunicación con la red CAN, memoria SD y valores de las variables analógicas o digitales, sea el caso. Como se mencionó anteriormente los comandos son enviados con la función *Serial.print* en la programación de Arduino y como el protocolo lo indica después de cada comando se imprimirá un retorno de carro, mediante la función *Serial.println.*

Por ejemplo para inicializar la pantalla se deberá utilizar la siguiente función, *Serial.println("@I20X4").* Como se observa el comando inicializa un LCD serial 20x4 de una manera sencilla.

Para el comando imprimir cadena de datos se utilizaría la misma sintaxis, *Serial.println("@E texto aquí")* sin embargo siempre se tendría que estar repitiendo está línea para imprimir mensajes en el LCD y además para visualizar el valor de variables es necesario utilizar otra función que se describirá más adelante, esto aumentaría el tamaño del código y no sería amigable con el usuario ya que son tareas repetitivas, por lo tanto, para hacer más fácil la programación se crearán unas funciones para simplificar la programación con un orden y que sea más clara e intuitiva para el usuario. Con estas funciones solamente se tendrá que mandar a llamar la función en el código principal para imprimir cadenas de datos, variables o ir a una posición en el LCD etc.

En las tablas 3.2 y 3.3 se muestran las funciones. En la primera columna se muestra el nombre de la función y en la segunda columna se muestra el código asociado a la función. Es importante mencionar que los parámetros de las funciones pueden ser modificados de acuerdo a la necesidad del usuario para los fines requeridos, el especialista puede modificar estas funciones de acuerdo al caso de prueba que vaya a realizar, sin embargo los valores fijados son funcionales para un número amplio de aplicaciones como se mostrara en el

capítulo de resultados. En la tercera columna se encuentra una descripción de la función donde se explica el tipo del valor que devuelve y la manera de llamarla en el código principal.

Función	Borrar LCD	Descripción
Borrar LCD	*void borrarLCD(void)* *{* *Serial.println("@B");* *}*	Borra el contenido del LCD, en la programación se mandará a llamar como: borrar LCD ().
Enunciado LCD	*void enunciadoLCD(char *enunciado)* *{* *Serial.print("@E");* *Serial.println(enunciado);* *}*	Permite imprimir una cadena de datos. En la programación se mandará a llamar como: enunciadoLCD(“Texto aquí”). Como se observa sigue respetando el protocolo.
Posición de cursor	*void posicionLCD(char renglon, char columna)* *{char array[5];* *Serial.print("@P");* *itoa(renglon,array,10);* *Serial.print(array);* *Serial.write(',');* *itoa(columna,array,10);* *Serial.println(array);* *delayLCD();* *}*	Coloca el cursor en la posición deseada para empezar a imprimir datos en el LCD. Está función utiliza la función itoa para convertir un número en una cadena de caracteres en una variedad de bases [5]. En el ejemplo se observa cómo el renglón y columna son convertidos en cadenas para ser utilizados en el comando. En la programación únicamente se llamará a la función posición LCD.

Tabla 3.2 Funciones para LCD serial.

Función	Borrar LCD	Descripción
Color	*VoidcolorLCD(char numero_color)* *{ char array[5];* *Serial.print("@L");* *itoa(numero_color,array,10);* *Serial.println(array);* *delayLCD();* *}*	Está función es válida si el LCD fuera RGB, sin embargo para un LCD monocromático se utiliza para fijar la iluminación del mismo. Se utiliza un valor de entre 0 y 19 para fijar el valor de intensidad, la iluminación o el color de la misma. Como se observa sigue la misma lógica que las funciones anteriores
Variables	*void numeroLCD(int variable)* *{char array[7];* *itoa(variable,array,10);* *enunciadoLCD(array);* *}*	Imprime el valor de una variable de tipo entero, se vuelve a utilizar la función *itoa*. Para utilizar esta función es necesario llamar a la librería *stdlib* al inicio del programa. El valor de variable es convertido a una cadena para poder usar la función enunciado LCD.

Tabla 3.3 Funciones para LCD serial.

Estas funciones ayudarán a optimizar el programa ya que solo se declararán una vez al final de la función cíclica del programa principal del IDE y serán llamadas cuando sea necesario.

En el anexo 5 se muestra el código de la implementación de un programa donde se imprime un enunciado y el valor que contiene la variable color para demostrar la efectividad de las funciones. El Arduino MEGA será el encargado de comunicarse con el LCD, por lo tanto, se ocupará un puerto serial de los 3 que

ofrece la tarjeta, quedando las funciones para el protocolo de la siguiente forma: Serial2.println(), donde el 2 indica que se define en el segundo puerto serie.

3.7 Conclusión

Se realizó la programación de un LCD serial con el IDE de Arduino utilizando comandos del protocolo UART.

Las funciones que se desarrollaron en este capítulo serán de mucha utilidad cuando se programen casos de prueba donde se deban mostrar valores de variables, o mensajes al especialista.

Utilizando el LCD serial se redujo el número de conexiones entre el Arduino y el LCD, dejando libre entradas digitales del Arduino que posteriormente serán utilizadas en la lectura de señales eléctricas.

CAPÍTULO 4

ACONDICIONAMIENTO DE SEÑALES ELÉCTRICAS.

4.1 Introducción

El uso de entradas multipropósito digitales y analógicas en el análisis de fallas es necesario en un dispositivo que automatice los procesos dentro de la empresa, ya que no tendría sentido su implementación si el dispositivo no contara con esta capacidad.

El dispositivo que se utiliza es una tarjeta Arduino con un microcontrolador de la familia ATmega, éste trabaja con voltajes TTL en sus entradas multipropósitos sin embargo no se puede usar directamente ya que los voltajes utilizados en la electrónica del automóvil son de 12 V para un uno lógico y 0 V para el cero lógico.

En el siguiente capítulo se presenta todo el diseño, la instrumentación de las entradas multipropósito hacia el microcontrolador y construcción de las placas PCB. Un aspecto relevante es la adaptación de salidas analógicas para cargas de potencia como lo son focos de los faros, motores DC que llegan a consumir 3 A de corriente. Por otra parte hay que considerar la integración de multiplexores I2C que dotaron al microcontrolador de más salidas digitales.

El número de entradas multipropósito con el que debe contar el dispositivo se describe a continuación:

1. Ocho entradas digitales.

2. Ocho salidas digitales.
3. Ocho entradas analógicas.
4. Cuatro salidas analógicas.

Sumando un total de 28 entradas multipropósito. Finalmente se presenta la integración con la programación de la interfaz LCD utilizando los conceptos del capítulo 3.

4.2 Voltaje analógico

El dispositivo deberá contar con salidas analógicas a 12 V y estas salidas deberán suministrar hasta 3 A para la carga.

Uno de los métodos comúnmente usados en control es utilizar DAC´s para que, a partir de un valor digital, transformarlo a un valor analógico de 0 V a 12 V. Esta aplicación solo es útil integrándole una etapa de potencia para el amperaje requerido.

La resolución de las salidas debe ser de 8 bits por lo tanto se utilizará el DAC0808 el cual ofrece esa característica [6], sin embargo por cada DAC se necesitarán 8 entradas multipropósito del Arduino. Este dispositivo ha utilizado 40 entradas multipropósito con las entrada digitales, salidas digitales, Shield CAN y LCD, por lo que ahora solo quedan 14 entradas de las 54 que tiene la tarjeta Arduino.

Se necesitarán 32 salidas digitales para variar el voltaje analógico con los DAC´s, por lo cual se tendrá que aumentar el número de estas salidas de la tarjeta Arduino utilizando registros de desplazamiento.

4.2.1 Registros de desplazamiento y el MCP23017

Los registros son circuitos secuenciales muy utilizados en la realización de sistemas digitales. Un registro es un bloque funcional que almacena o registra información binaria durante cierto tiempo. Un registro de *n* bits está formado por un conjunto de *n* flip-flops, donde cada uno almacena un bit de la información.

Actualmente existen registros que utilizan diferentes tipos de comunicación como lo son I2C o SPI que en capítulos anteriores se ha explicado cómo funcionan.

La configuración de las entradas y salidas son lo que determinan el tipo de registro. El tipo de registros que son de interés para esta aplicación es el SIPO (*Serial Input Parallel Output)*. En la figura 4.1 se muestra esta configuración.

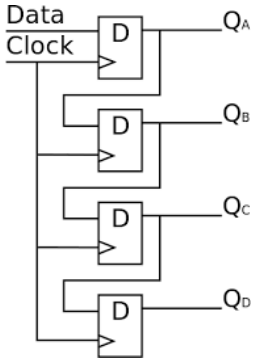


Figura 4.1 Registro tipo SIPO.

El MCP23017 consiste en 2 registros con una interfaz serial por medio del protocolo I2C y cuenta con 2 puertos (A y B) de 8 bits del tipo SIPO [7].

La conexión se muestra en la figura 4.2, donde se observan las resistencias de pull up de 4.7 KΩ, las líneas SCL y SDA van conectadas a los pines 20 y 21 del Arduino. Los pines de dirección son los números 15, 16 y 17 del integrado. En la misma figura se observa que estos pines están conectados a tierra, por lo tanto la dirección es 0x20 en el protocolo I2C; esto es muy útil ya que se pueden tener más registros y aumentar las salida del Arduino.

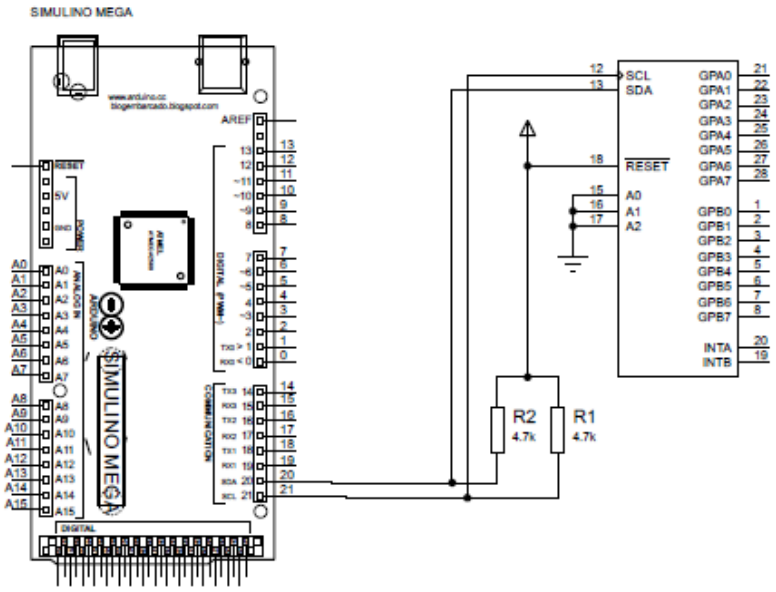


Figura 4.2 Conexión entre Arduino y MCP23017.

Si estos pines se conectan todos a voltaje positivo se tendrá como dirección 0x27, este es el valor que tendrá nuestro segundo integrado ya que se necesitan aun 16 salidas digitales más.

Para programar en el IDE de Arduino se utilizará la biblioteca *"wire"* con la cual es posible comunicarse con dispositivos I2C utilizando funciones sencillas y amigables para el programador.

1. *Wire.begin:* Inicializa la comunicación con el BUS I2C y la biblioteca.
2. *Wire.write:* Escribe los datos en un dispositivo esclavo en respuesta a una petición de un maestro, o colas de bytes para la transmisión de un maestro al dispositivo esclavo.
3. Wire.endTransmission: Finaliza una transmisión a un dispositivo esclavo y transmite los bytes que se han puesto en cola.

En el siguiente código se inicializa la comunicación I2C, se escribe la dirección del registro al que se va a acceder y se definen todos sus pines como salidas, ya que el integrado no está limitado a escribir, también se pueden leer datos.

Wire.beginTransmission(0x20); Dirección del dispositivo en el BUS

Wire.write(0x00); // Dirección del Registro A.

Wire.write(0x00); // Todo el registro como salida.

Wire.endTransmission(); Finaliza la comunicación.

Este código solamente se ejecutará una vez por lo que se incluye en el void setup. Los datos se enviarán con el siguiente código:

Wire.beginTransmission(0x20);

Wire.write(0x12); // Dirección de puerto A.

Wire.write(¿?); // Valor a enviar.

Wire.endTransmission();

Se vuelve a escribir la dirección del registro A y después el valor que se reflejará a la salida del registro y se finaliza la comunicación. Para enviar los datos este código se inserta en la función void loop.

Para el puerto B se utiliza la misma sintaxis, tan solo se cambia la dirección *0x20* a *0x27* con sus pines de control a voltaje positivo.

4.2.2 Conversión de valores digitales a analógicos

Con el MCP23017 se tiene la parte de los valores que irán a los DAC. Los convertidores DAC 0808 necesitan un voltaje negativo para funcionar, sin embargo el dispositivo solamente se podrá alimentar de voltaje positivo, por lo tanto se tendrá que convertir el voltaje positivo a negativo. Los integrados que permiten hacer esto se llaman convertidores DC/DC; entre sus diversas aplicaciones también existen la de fuente negativa, un voltaje positivo lo invierte y disminuye la corriente pero aun así puede alimentar otros integrados como OPAMPS o DAC´s que requieren este tipo de voltaje. EL integrado ICL7662 es un convertidor DC/DC de tecnología CMOS, el cual puede invertir hasta -18 V utilizando 2 capacitores de 10 µF [8].

La conexión entre el Arduino, el MCP23017, el circuito típico del DAC 0808 [9] y el ICL7662 se muestra en la figura 4.3.

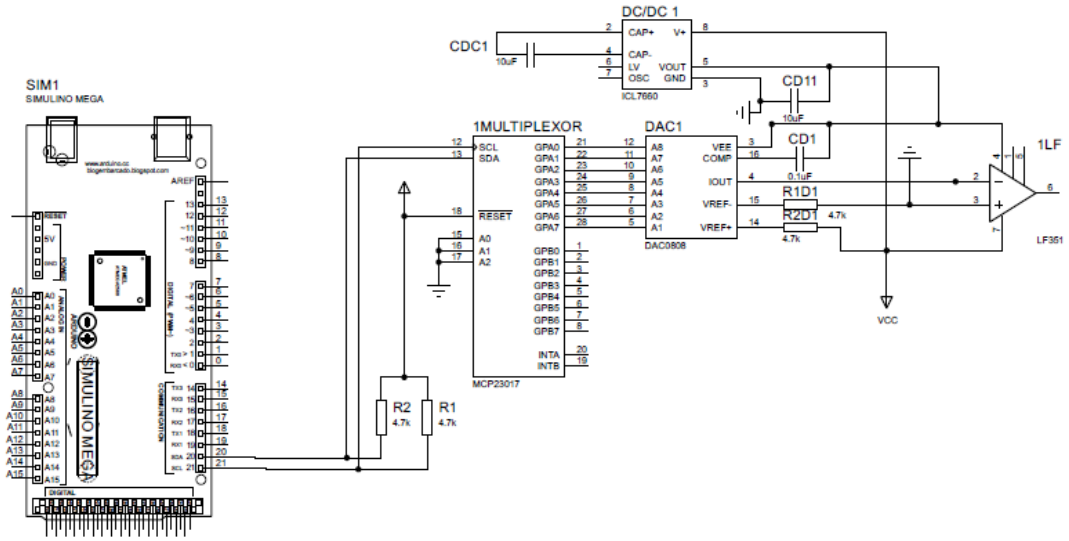


Figura 4.3 Integración de circuitos MCP23017, ICL7662 y DAC0808.

Los pines del integrado MCP23017 se conectan con sus respectivos pines del DAC del bit más significativo al menos significativo. El DAC convierte el valor digital en corriente, el OPAMP a la salida se encarga de convertirlo en voltaje.

El ICL7662 alimenta al OPAMP y al DAC en sus respectivos voltajes negativos, a pesar de que existen configuraciones de OPAMS donde no es necesario utilizar un voltaje negativo, los OPAMPS que pueden ser reemplazados a la salida del DAC no pueden ser implementados con estas configuraciones.

Las resistencias que se utilizan son de 4.7 KΩ como lo marca el fabricante para la referencia positiva y negativa. La referencia negativa se manda a tierra ya que no se ocuparán voltajes negativos. Finalmente se podrá variar el voltaje con la pérdida correspondiente del DAC y del OPAMP.

4.2.3 Etapa de potencia

La corriente que un OPAMP puede suministrar es de unos pocos mA. Está corriente es ideal en aplicaciones de control sin embargo se requiere alimentar

motores o focos del automóvil que demandan demasiada corriente, por lo que si se conecta directamente la carga se dañaría el OPAMP.

La mayor parte de fuentes variables DC de potencia utilizan integrados LM350 ya que alimentan cargas que demandan 3 A garantizado por el fabricante y soportan hasta un máximo de 5 A [10]. El voltaje a la salida varía dependiendo el valor del voltaje resultado de un divisor en la terminal de ajuste del integrado, este voltaje varía por pocos mV con respecto al de salida.

Partiendo de esa idea, se aplicará el voltaje del OPAMP al pin de ajuste del LM350, obteniendo a la salida el mismo voltaje pero con capacidad de alimentar a las cargas a utilizar. En la figura 4.4 se anexa el circuito final donde se agrega el integrado LM350.

Este circuito se repite para las 4 salidas analógicas, el código del programa se muestra en el anexo 6.

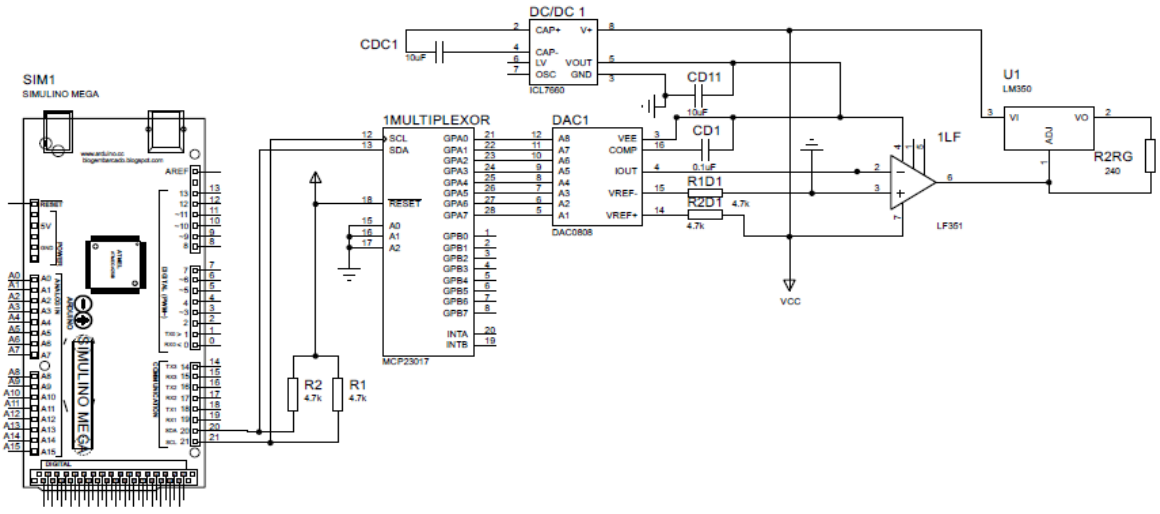


Figura 4.4 Circuito para salida analógica con consumo de hasta 3 A.

La potencia disipada por los integrados LM350 se convierte en calor; el integrado se mantendrá disipando esta potencia por horas por lo que es necesario implementar disipadores de calor antes de que entre su circuito de protección y

detenga su función. Existe un método para el cálculo del tamaño de disipador para el regulador de voltaje.

Se puede hacer una analógica con circuitos eléctricos y utilizar la ley de ohm al igual que en los circuitos eléctricos.

Intensidad	*Calor (W)*
Voltaje	*Temperatura(°C)*
Resistencia	*Resistencia térmica (°C/W)*
V=IR	*T=W*R*

Tabla 4.1 Equivalencia térmica y eléctrica.

En la tabla 4.1 se muestra el circuito equivalente, donde la resistencia térmica es igual a la asociación de resistencias en serie, se suman los valores de cada una de ellas y lógicamente, cuanto mayor es la resistencia térmica, mayor dificultad para el flujo de calor.

En la ecuación 4.1 se observa la ecuación final [11].

$$T = T_j k - Ta = W(R_{jc} + R_{cd} + R_{da}) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (4.1)$$

Dónde:

Tj = Temperatura de la unión.

Ta = Temperatura ambiente.

R_{jc} = Resistencia térmica unión-cápsula.

R_{cd}= Resistencia térmica cápsula-disipador.

R_{da}= Resistencia térmica disipador-ambiente.

k =0.5 para un diseño normal con temperatura moderada.

k = 0.6 para economizar en tamaño de disipador.

k = 0.7 cuando el disipador permanezca en posición vertical y en el exterior.

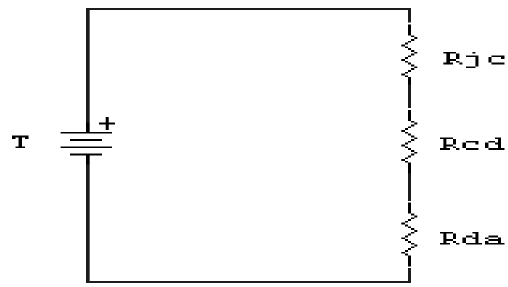


Figura 4.5 Circuito equivalente.

Para el cálculo de la resistencia disipador ambiente, se utiliza la ecuación 4.2.

$$\frac{T_j k - Ta}{W} - \left(R_{jc} + R_{cd}\right) = R_{da} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots..\ldots..(4.2)$$

Los demás datos se pueden encontrar en la hoja de datos del fabricante.

Tj = 125°C

Ta = 25°C

R_{jc} = $3°C/W$

R_{cd}= Se toma el valor entre 0.5°C - 1°C sin mica aislante y con pasta conductora entre el disipador y el integrado.

$W(V * I)$= La potencia a disipar. La corriente es de 3 A con un voltaje de 12 V por lo tanto tenemos 36 W.

Sustituyendo los valores en la ecuación 4.3**.**

$$R_{da} = \frac{125° * 0.7 - 26°C}{36W} - \left(3°\frac{C}{w} + 1°C\right) = -2.29 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots.\ldots\ldots\ldots.(4.3)$$

Como se observa en la ecuación 4.3 el valor de resistencia es negativo por lo tanto no es posible enfriar el integrado con disipadores de aluminio, si el valor hubiera sido positivo el valor de resistencia se asocia a un disipador de aluminio de determinadas dimensiones. Por tal motivo se utilizará un enfriador por aire y

aparte un disipador de aluminio en el integrado, para disipar la mayor parte de aire caliente.

El circuito final se muestra en el anexo 7 y la placa PCB diseñada en el programa ISIS Proteus se muestra en el anexo 8.

4.3 Lectura de voltaje analógico

Como se mostró anteriormente una señal analógica es una magnitud que puede tomar cualquier valor dentro de un intervalo –Vcc a + Vcc**.** Por ejemplo es posible variar el voltaje y fijar un valor de 2.34 V entre 0 V y 12 V o cualquier otro valor dependiendo la resolución del dispositivo. Sin embargo por lo general en los autómatas las entradas analógicas son más escazas, más lentas y más caras que las entradas digitales.

El proceso anterior fue asignar un valor digital desde la programación de Arduino a uno analógico, ahora para leer los voltajes exteriores es necesario invertir el proceso, a partir de un valor analógico que se leerá con el Arduino se le asignará un valor digital con una resolución de 8 bits.

Cuando se tienen que procesar señales analógicas mediante sistemas digitales se precisan de unos circuitos denominados ADC (Analog to Digital Converter). En la figura 4.6 se muestra una representación de un ADC. De esta manera el Arduino obtiene la información digital correspondiente a la señal analógica

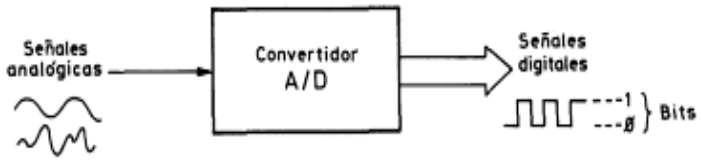


Figura 4.6 Convertidor de señales analógicas a digitales.

El Arduino MEGA ofrece 16 entradas analógicas con su respectivo ADC que proporciona la medición codificada de un valor digital de hasta 10 bits de resolución utilizando funciones que ofrece el IDE de Arduino.

- AnalogRead: Esta función lee el valor del pin analógico especificado.

- AnalogReference: Modifica el rango de entrada y la resolución de los pines analógicos.

El proceso tarda unos 100 microsegundos para leer el valor analógico de las entradas por lo que la tasa máxima de lectura es de diez mil veces por segundo. Si el pin de entrada analógica no está conectada a nada, el valor devuelto por la función analogRead fluctuará en una base a una serie de factores.

El voltaje de lectura permitido es de 0 V a 5 V, sin embargo se desea leer un voltaje entre 0 V y 12 V este voltaje dañaría al microcontrolador.

Se implementará un divisor de voltaje con las siguientes características, el voltaje máximo al que estará diseñado será de 15 V; este voltaje es el máximo que podría suministra el automóvil cuando se prende, el cual tendrá que ser equivalente a 5 V que es el que leerá el Arduino y en la programación se mapearán los valores para asignar su valor correspondiente. En la figura 4.7 se muestra el circuito de un divisor de voltaje, $Vin = 15\,V$ $y\,Vout$ el voltaje deseado.

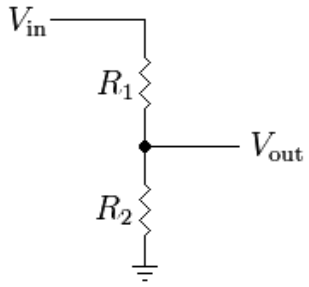


Figura 4.7 Divisor de voltaje.

El cálculo para el divisor de voltaje se muestra en la ecuación 4.4.

$$Vout = Vin \frac{R_2}{R_1 + R_2} \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots. (4.4)$$

Se fija R_1 a 2.4 KΩ y se despeja R_2 con $\mathrm{Vin} = 15\ \mathrm{V}$ y $\mathrm{Vout} = 5\ \mathrm{V}$. Se sustituyen los valores en la ecuación 4.5.

$$R_2 = \frac{Vout * R_1 * Vin}{Vin^2 - Vout * Vin} = \frac{5V * 2400\Omega * 15V}{15V^2 - 5V * 15V} = 1200\Omega \dots\dots\dots\dots\dots\dots\dots\dots\dots. (4.5)$$

El valor de la resistencia 2 es de 1200 Ω, el número de entradas analógicas del dispositivo es de 8 por lo tanto se tendrán que hacer 8 divisores de voltaje. En la figura 4.8 se muestra el circuito con los 8 divisores de voltaje.

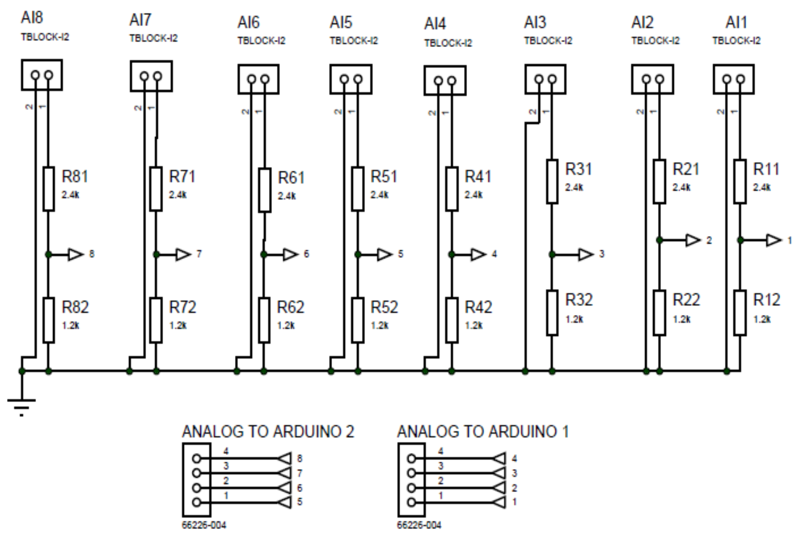


Figura 4.8 Circuito de entradas analógicas.

Este voltaje es enviado posteriormente a los pines analógicos del Arduino MEGA del pin A8 al pin A9.

4.4 Lectura de voltaje digital.

Las terminales de Arduino, por defecto, están configuradas como entradas, por lo tanto no es necesario definirlas en el caso de que vayan a funcionar como entradas. Los pines configurados tienen una alta impedancia, lo anterior implica que con poca corriente es suficiente para cambiar de estado del Arduino.

Los pines tienen a nivel interno una resistencia de 20 KΩ que se puede activar o desactivar mediante software. Si el pin de Arduino se encuentra activo con sus resistencias internas y no se conecta a nada este fácilmente cambiará su estado debido al ruido eléctrico o al cambio del pin cercano.

La manera de programar estas resistencias es: *pinMode(pin,INPUT),* donde pin corresponde al número del pin del Arduino, de esta manera queda configurado el modo de trabajo de un pin pudiendo ser INPUT (entrada) u OUTPUT (salida).

Un cortocircuito en los pines digitales de Arduino provocará una corriente elevada que puede dañar o destruir el chip ATmega.

Los estados digitales de la lectura ya sea alto (HIGH) y bajo (LOW) se obtienen con la función *digitalRead(pin)* del pin de interés y se asigna a una variable o una constante.

Los voltajes permitidos para Arduino son de 0 V para un valor bajo y 5 V para un valor alto, teniendo como máximo 5.5 V. Sin embargo los voltajes digitales en el automóvil son de 0 V y 12 V, estos voltajes si son conectados directo al Arduino, dañaría el microcontrolador de la tarjeta.

Para acondicionar estas señales se utilizarán reguladores de voltaje LM7805 que tendrán como función regular los 12 V que se tienen como voltaje alto a 5 V en el Arduino.

En la figura 4.9 se muestra el circuito para las entradas digitales, como se observa por la terminal Vin se aplicaría el voltaje de 12 V y a la salida se obtendría un voltaje máximo de 5 V. Este voltaje es el que se aplicaría a los pines del

Arduino MEGA. El circuito que integra a las entradas digitales y entradas analógicas se muestra en el anexo 9 y la placa PCB diseñada en el programa ISIS Proteus se muestra en el anexo 10, donde se realizó el circuito de las entradas digitales y analógicas en la misma placa.

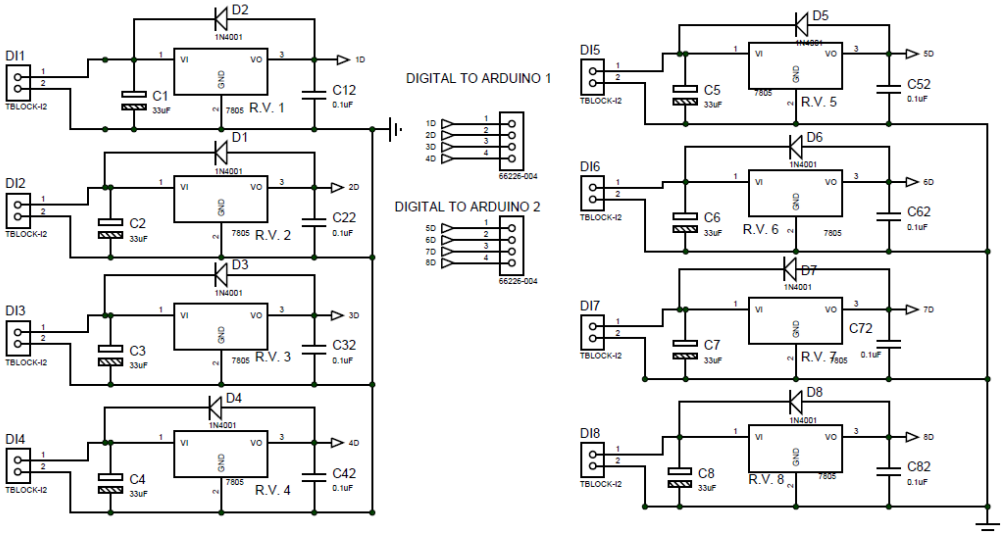


Figura 4.9 Circuito para 8 entradas digitales.

4.5 Salidas Digitales

Los pines del Arduino MEGA configurados como salidas se encuentran en un estado de baja impedancia y pueden proporcionar 40 mA de corriente a otros dispositivos y circuitos. Esta corriente es suficiente para alimentar integrados o diodos led, pero no es lo suficientemente grande como para alimentar cargas de mayor consumo como relés, solenoides o motores, ya que si se alimentan directamente dañarían al microcontrolador.

Para definir los pines del Arduino como salidas se utiliza la función *pinMode(pin,OUTPUT)* y para poder activar estos pines es la digitalWrite(pin, HIGH) donde pin indica el número de pin definido como salida y posteriormente el que se activará; este pin puede ser definido como constante para facilitar la programación, quedando de la siguiente manera.

#define rele1 41; // Define a rele1 con valor 41

```
void setup()
{
pinMode(rele1, OUTPUT); // configura rele1 (pin 41) como salida
}
void loop()
{
digitalWrite(rele1, HIGH); }  // Activa rele1
```

Como se observa la progresión de las salidas digitales es amigable con el usuario. Las salidas digitales como se mencionó anteriormente son de un voltaje de 5 V y de baja corriente por lo que para la aplicación no es útil. Debido a esto, se anexará una etapa de relevadores que permitirá utilizar voltajes y corrientes mayores.

Se utilizará un módulo de relevadores para poder activar salidas digitales a 12 V; la placa utilizada contiene 8 relevadores con sus respectivos circuitos. En la figura 4.10 se muestra el esquemático.

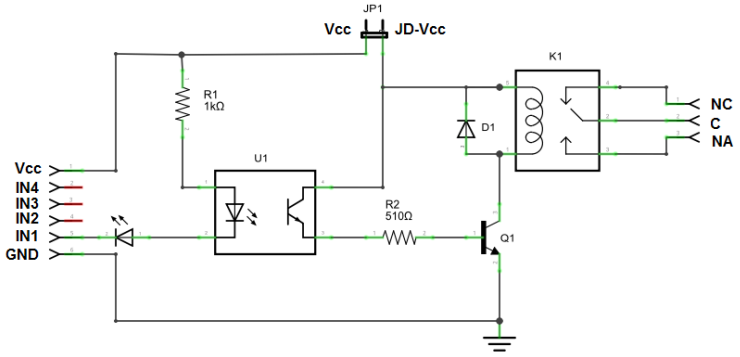


Figura 4.10 Circuito para conexión de un relevador.

El funcionamiento del circuito es simple, la entrada IN1 está conectada al cátodo del diodo del optoacoplador FL817C a través del led indicador. El ánodo del diodo del optoacoplador se conecta a una resistencia de 1 kΩ y a VCC, formando un circuito serie por el cual circula la corriente cuando la entrada está a un nivel BAJO y no circula si la entrada está a un nivel ALTO.

El transistor del optoacoplador tiene su colector a JD-Vcc y su emisor conectado a Q1 a través de una resistencia de 510 Ω. Cuando el optoacoplador

permite el paso de corriente se satura el transistor permitiendo el paso de la corriente a través de la bobina del relevador cerrando el contacto de la terminal C a NA.

Por lo tanto en la programación del IDE de Arduino cuando se manda una señal en alto el relevador no se energizará y con una señal en bajo se energizará. El programa para activar los relevadores se muestra en el anexo 11.

4.6 Conclusión

Se realizó el acondicionamiento de las señales eléctricas multipropósito quedando de la siguiente manera:

1. Ocho entradas digitales capaces de soportar un valor de 15 V por seguridad del dispositivo.
2. Ocho salidas digitales con un voltaje de 12 V.
3. Ocho entradas analógicas con una resolución de 10 bits con un rango de 0 V a 15 V.
4. Cuatro salidas analógicas, con una capacidad de entregar a la carga hasta 3 Amperes y un voltaje variable de 0 V a 12 V.

Se realizó el procesamiento de todas las señales con un Arduino MEGA utilizando registros de desplazamiento con el protocolo I2C para aumentar el número de puertos del Arduino.

CAPÍTULO 5

RESULTADOS EN AUTOMÓVIL GOLF A7

5.1 Introducción

En el presente capítulo se describe la realización de un caso de prueba utilizando la mayor parte de entradas multipropósito del dispositivo, leyendo y grabando mensajes del BUS CAN de un automóvil de plataforma MQB a una velocidad de 500 kb/s. Al mismo tiempo se muestran mensajes en el LCD de inicialización del sistema y valores de las entradas multipropósito.

5.2 Elaboración de un caso de prueba libre

Un caso de prueba libre es una rutina de acciones al azar que no se encuentra en un catálogo de pruebas establecidas por el fabricante pudiendo llegar a generar una falla en la unidad de control.

Independientemente de las acciones que realice el dispositivo en el caso de prueba se estarán leyendo mensajes en los 2 canales del BUS CAN del automóvil con su respectiva impresión de tiempo.

El sistema iniciará con la siguiente secuencia de eventos que conforma nuestro caso de prueba libre:

1. El LCD comenzará mostrando los siguientes mensajes: "*Inicializando tarjeta SD1"*, si la tarjeta micro SD del Arduino MEGA se inicializa correctamente aparecerá el mensaje "*Se ha inicializado SD1"* y comenzará a inicializar la

tarjeta del Arduino UNO, en caso contrario el mensaje será "*No se inicializó la SD1"*. Se vuelven a repetir los mismos mensajes para la memoria del Arduino UNO.

El mensaje siguiente en el LCD será "*Inicializando comunicación CAN1"*, donde se carga la configuración correspondiente en el Arduino MEGA para poder efectuar la comunicación con el Shield CAN, en caso de ser exitosa esta configuración la comunicación mostrará el mensaje *"Conexión correcta de CAN1"*, de no ser así mostrará "*Imposible conectarse con CAN1".* El mensaje siguiente será el de la inicialización del Shield CAN con el Arduino UNO, en el LCD se desplegará el mensaje "*Inicializando comunicación CAN2"*, posteriormente el mensaje "*Conexión correcta de CAN2".* Después de estos mensajes el Arduino UNO estará listo para comenzar a leer mensajes del BUS CAN, en caso de que el Arduino haya tenido un error y no pudiera enlazar una comunicación con el Shield CAN o se encuentre apagado el LCD desplegará "*Imposible conectarse con CAN2".*

2. Hasta este momento los dos Arduinos ya han comenzado a leer mensajes del BUS CAN y almacenado los mensajes en la memoria SD. El usuario podrá percatarse de ello ya que se utilizaron los LEDs con el número 7 de ambos Shield que estarán encendidos. Es importante mencionar que los anteriores mensajes solamente inicializan las tarjetas y la memoria SD de ambos Shield y no garantizan aún la conexión con el automóvil.

   Ahora comenzará el caso libre. Se empezará por activar cuatro salidas digitales, las dos primeras se activan cada medio segundo y las dos últimas cada segundo.
3. Se activarán la 2 salidas analógicas que comenzarán con un valor de 2.4 V hasta alcanzar el valor de 11.5 V y se reiniciará el valor.
4. Se leerán todas las entradas analógicas del Arduino y se imprimirán los valores después de cada lectura de los mensajes CAN en la memoria SD, es decir, después de leer un mensaje hay un retorno de carro y se imprime

el valor de las entradas analógicas separadas por una coma, además el valor de tres señales se mostrarán en el LCD.

5. La robustez del sistema debe garantizar que el proceso se esté realizando indefinidamente o hasta que se le detenga ya sea por hardware o software.

5.3 Estuche del dispositivo

El diseño del estuche que contiene los 2 Arduinos con sus respectivos Shield CAN, las placas PCB de las entradas multipropósito, el LCD y un ventilador a 12 V que enfría la placa de la parte de la electrónica de potencia se realizó con el Software Solid Works.

En la figura 5.1 y 5.2 se puede observar la parte frontal y posterior. El diseño se realizó en Solid Works para posteriormente ser mandado a cortar con una maquina CNC con cortador láser.

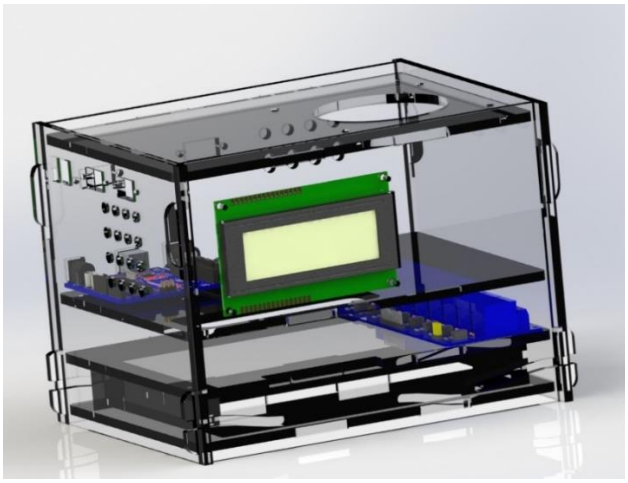


Figura 5.1 Vista de parte frontal del dispositivo.

Figura 5.2 Vista trasera del dispositivo.

En la figura 5.3 se muestra el dispositivo ya ensamblado. Se utilizaron conectores banana tipo hembra para facilitar la conexión de las salidas digitales y analógicas, conectores DB15 para las entradas analógicas y digitales y finalmente conectores DB9 para los conectores del Shield CAN.

Del primer conector DB15 del pin número 1 al número 8 son entradas digitales y del número 9 al número 15 son tierra común, de igual forma para el segundo conector DB15 correspondiente a las entradas analógicas del pin número 1 al número 8 son entradas analógicas y del número 9 al número 15 son tierra común. Se utilizaron estos conectores para no saturar de conectores tipo banana al estuche y causar confusión con las salidas digitales y analógicas, además cada conector tiene rotulado su función con una etiqueta.

Las salidas digitales se muestran en la parte derecha del dispositivo y las analógicas en la parte inferior del estuche con sus conectores tipo banana. Es importante conocer dónde se encuentran todas las salidas multipropósito y conectores CAN para realizar la implementación del caso de prueba. El estuche contiene tres niveles, en el primer nivel se encuentra la placa con los multiplexores y DAC´s, en el segundo nivel se ubica la placa con la electrónica de las entradas analógicas y digitales, mientras que en el último nivel se encuentran los Arduinos con los Shield CAN y la placa con la electrónica de potencia.

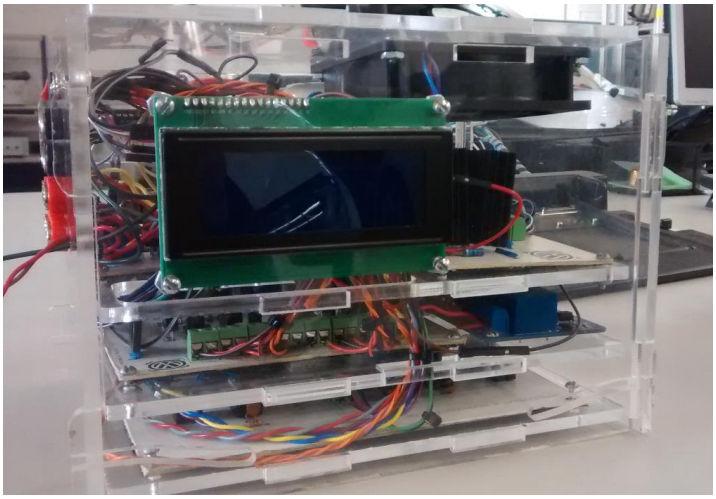


Figura 5.3 Dispositivo ensamblado.

5.4 Implementación del caso de prueba

En las figuras 5.4 a 5.6 se muestran los mensajes de inicialización del dispositivo en el LCD.

Figura 5.4 Mensaje de inicialización.

Figura 5.5 SD del Arduino MEGA Inicializada.

Figura 5.6 Tarjeta SD del Arduino UNO inicializada.

Las figuras anteriores muestran los mensajes en el LCD de la correcta inicialización de las memorias SD.

Las figuras 5.7 y 5.8 corresponden a los mensajes de la exitosa comunicación del Arduino MEGA con el Shield CAN.

Figura 5.7 Conexión correcta entre Shield CAN y Arduino MEGA.

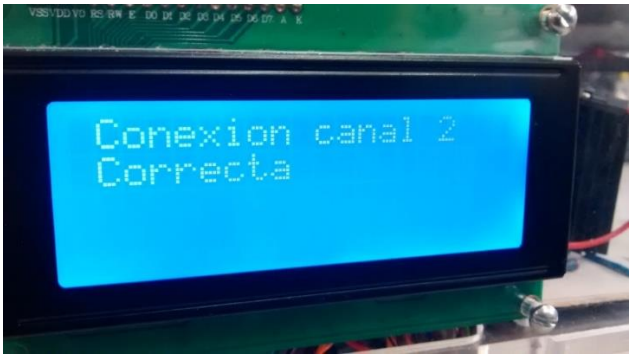


Figura 5. 8 Conexión correcta entre Shield CAN y Arduino UNO.

En la figura 5.9 se observa el archivo de texto generado por el Arduino MEGA con los mensajes CAN y los valores de las entradas analógicas; es importante mencionar que el dispositivo seguirá escribiendo los datos de los mensajes CAN en el mismo archivo de texto a menos que se le indique en la programación la creación de un nuevo archivo en el próximo caso de prueba.

En la figura 5.9 se muestran los datos de los mensajes CAN del archivo de texto del Arduino UNO con su impresión de tiempo y fecha de creación del archivo.

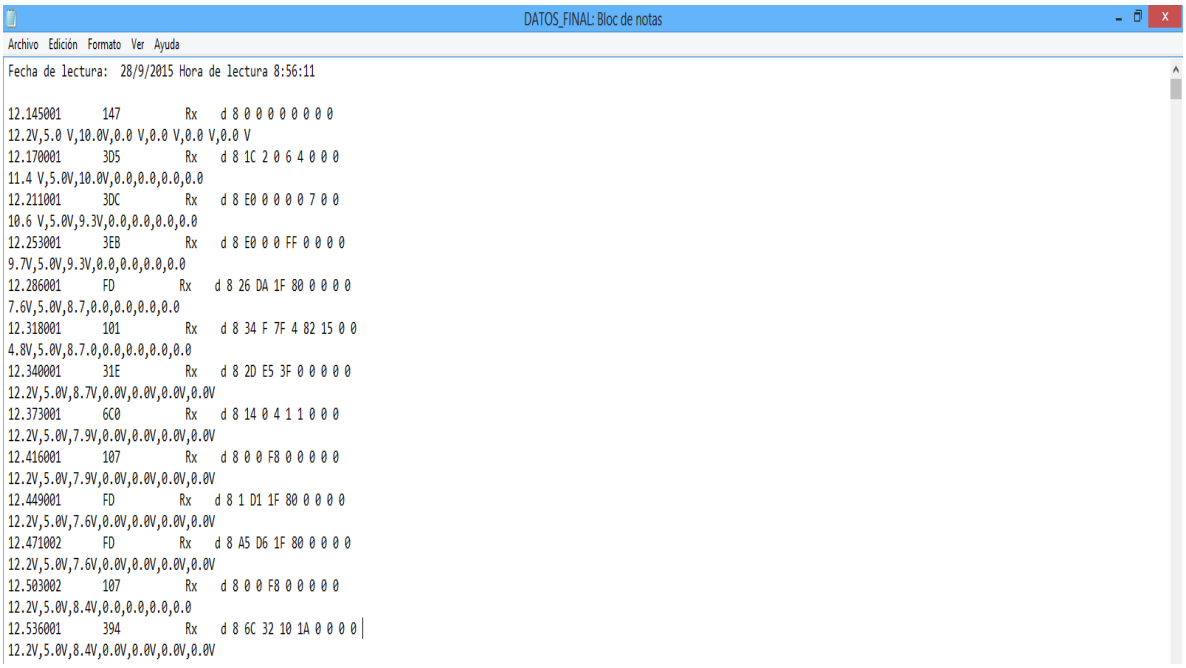


Figura 5.9 Archivo generado por el Arduino.

La figura 5.10 muestra el LCD imprimiendo el valor de las variables analógicas. Debido al bucle en el que está la impresión de las señales eléctricas en la programación del LCD el cambio se aprecia rápidamente. Se utiliza el voltaje generado del puerto USB de una computadora para alimentar una entrada analógica y el de una fuente variable para las otras 2 entradas. Se observa en el archivo de texto que no hay un retraso en las lecturas de los valores analógicos grabados en la memoria SD.

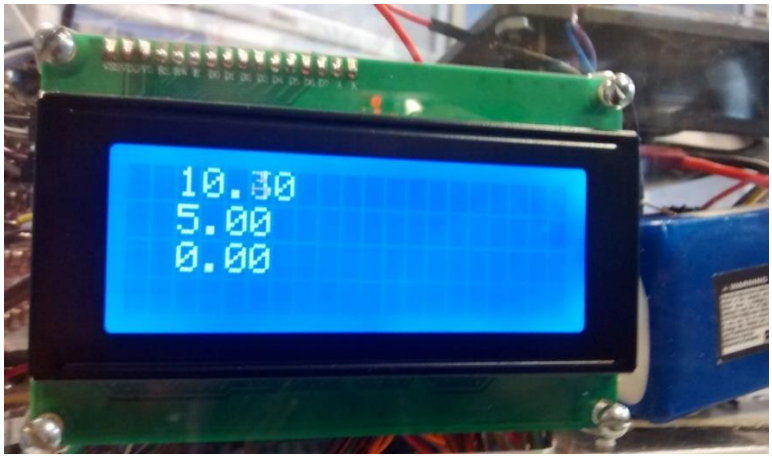


Figura 5.10 Valor de voltajes analógicos.

Las salidas digitales y analógicas son mandadas a focos no montados en el automóvil pero si pertenecientes a los faros del GOLF, debido a que no se tiene una interfaz de acceso al arnés del automóvil.

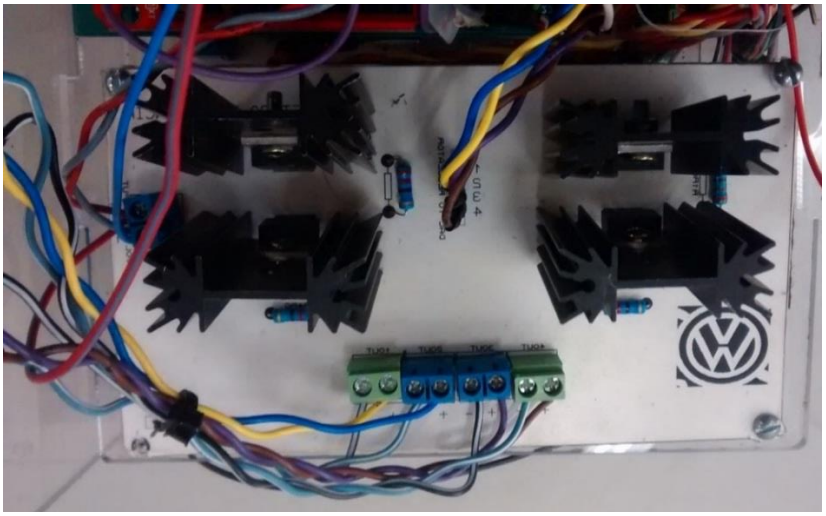


Figura 5.11 PCB de la electrónica de potencia.

La figura 5.11 muestra la placa PCB correspondiente a la electrónica de potencia del dispositivo.

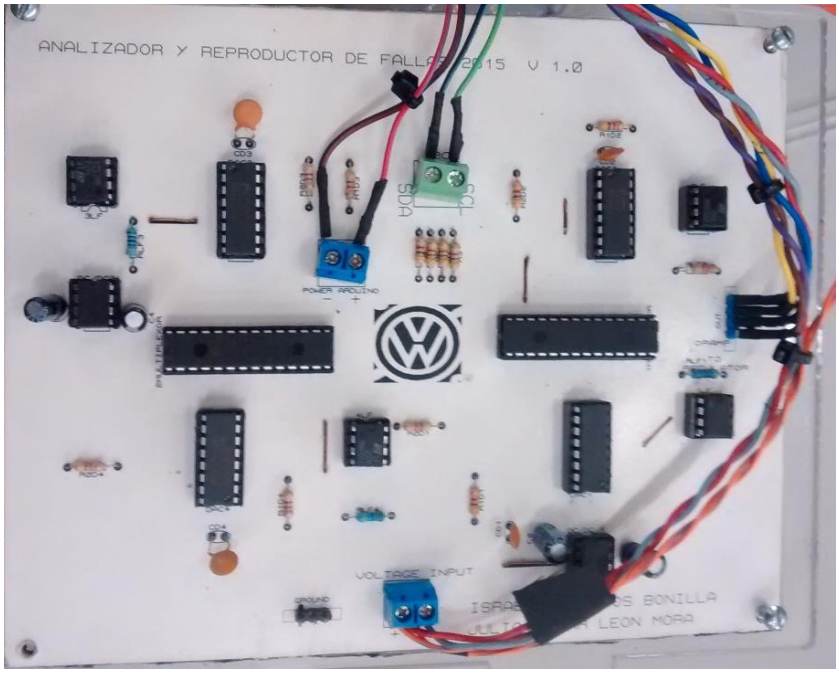


Figura 5.12 PCB correspondiente a las salidas analógicas.

La figura 5.12 muestra la PCB correspondiente a las salidas analógicas descritas en el capítulo 4.

La figura 5.13 muestra la parte superior de los Shield CAN conectados con un cable DB9 y éste a su vez conectado a una interfaz del automóvil que se conecta a la red CAN.

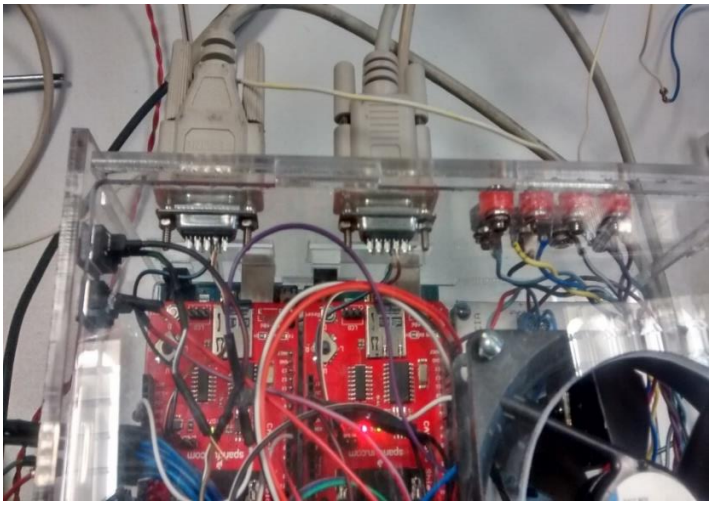


Figura 5.13 Shields CAN del Arduino MEGA y Arduino UNO.

5.5 Conclusión

Se construyó un estuche que contiene los Shield CAN con los Arduinos y las PCB correspondientes a la instrumentación de las señales analógicas y digitales, con sus correspondientes conectores para la lectura o recepción de las señales eléctricas y de CAN.

Se realizó un caso de prueba libre donde se integró la activación, lectura y grabación de las señales eléctricas y CAN verificándose su correcto funcionamiento.

Se muestra evidencia de los resultados obtenidos durante el caso de prueba generando correctamente el archivo de texto con toda la información de las señales eléctricas y de CAN.

CAPÍTULO 6

CONCLUSIONES

Se realizó un dispositivo capaz de obtener mensajes de la red CAN de un automóvil con 2 canales de lectura CAN High Speed, con velocidad configurable de lectura a 100 kb/s, 125 kb/s y 500 kb/s, aplicable en cualquier plataforma de los automóviles de la marca Volkswagen. Es un proyecto de impacto industrial ya que brinda una solución que resuelve un problema de la gerencia de desarrollo técnico electrónico en el proceso de análisis de fallas de las unidades de control.

Se disminuye el tiempo en encontrar una falla ya que con la programación adecuada realiza acciones que un trabajador no podría reproducir como lo es activar actuadores en ventanas de tiempo de milisegundos; además brinda una solución económica y funcional en un dispositivo con un costo inferior a un datalogger comercial, además integra entradas multipropósito digitales y analógicas, donde las salidas analógicas presentan una excelente característica debido a que pueden alimentar actuadores que necesiten hasta más de 3 Amperes como lo son: motores eléctricos de limpia parabrisas, ventanas, compuertas de clima, espejos, focos de faros y calaveras. Esta característica permite realizar cualquier caso de prueba que involucre los actuadores anteriores ya que son mayoría en las redes de confort del automóvil.

Además garantiza un registro confiable de los valores de las señales eléctricas hacia el dispositivo y en el automóvil, así como la información de los mensajes CAN con una impresión de tiempo. La información que genera el dispositivo es transferida posteriormente a un software de análisis, ya que el formato en que genera la información es el correcto para su procesamiento y así es posible conocer en caso de una falla cuál es la unidad de control que genera el problema.

El dispositivo no depende de una computadora ya que cuenta con un LCD que permite la interacción con el usuario de una manera amigable con el Shield CAN y visualiza mensajes del funcionamiento del dispositivo o valores de las señales eléctricas.

Se realizó un caso de prueba libre en un automóvil Golf A7 donde se verificó su correcto funcionamiento, éste tipo de casos de prueba son fundamentales ya que realiza acciones aleatorias en los actuadores que no están documentadas por la empresa y se comprueba el correcto funcionamiento de las unidades de control. Una de las principales funciones del dispositivo será reproducir estos casos de prueba aleatorios mientras no exista una falla reportada de una unidad de control.

Bibliografía

[1] J. E. Domínguez, Redes de comunicación de datos (Circuitos eléctricos auxiliares del vehículo), Editex, 2012.

[2] UDLAP consultores, Introducción a redes automotrices, Puebla, 2014.

[3] C. Steve, «TEXAS INSTRUMENTS,» Julio 2008. [Online]. Available: http://www.ti.com/lit/an/sloa101a/sloa101a.pdf. [Accessed 12 Octubre 2015].

[4] M. T. Inc, «Microchip,» 28 Julio 2003. [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21801G.pdf. [Último acceso: 17 Octubre 2015].

[5] Arduino Genuino, «Arduino,» 30 Enero 2015. [En línea]. Available: http://playground.arduino.cc/Code/PrintingNumbers. [Último acceso: 25 Septiembre 2015].

[6] T. Instruments, «TEXAS INSTRUMENTS,» 9 Mayo 1999. [En línea]. Available: http://www.ti.com.cn/cn/lit/ds/symlink/dac0808.pdf. [Último acceso: 4 Noviembre 2015].

[7] Microchip, «Microchip Technology,» Febrero 2007. [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21952b.pdf. [Último acceso: 03 Noviembre 2015].

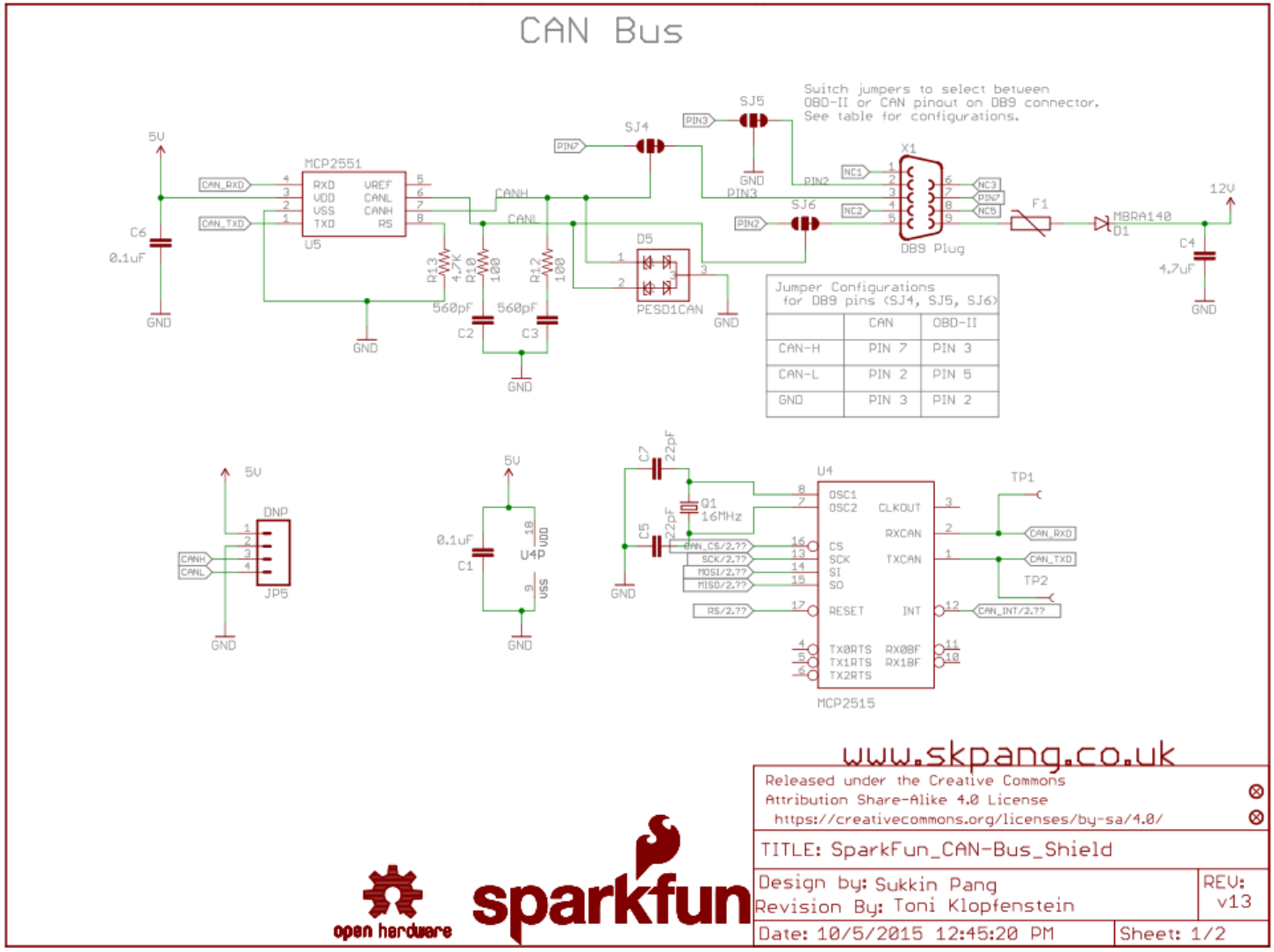
[8] M. T. Inc, «Microchip,» 28 Julio 2003. [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21468B.pdf. [Último acceso: 04 Septiembre 2015].

[9] T. Instruments, «TEXAS INSTRUMENTS,» 9 Mayo 1999. [En línea]. Available: http://www.ti.com.cn/cn/lit/ds/symlink/dac0808.pdf. [Último acceso: 4 Noviembre 2015].

[10] T. INSTRUMENTS, «TEXAS INSTRUMENTS,» Marzo 2013. [En línea]. Available: http://www.ti.com/lit/ds/symlink/lm350a.pdf. [Último acceso: 1 Octubre 2015].

[11] F. D. Trujillo, A. Pozo y A. Triviño, «Open Course de Universidad de Murcia,» 2011. [En línea]. Available: http://ocw.uma.es/ingenierias/electronica-de-potencia/ejercicios-proyectos-y-casos-1/calculo_de_disipadores.pdf. [Último acceso: 12 Octubre 2015].

Anexo 1


CAN Bus
Switch jumpers to select between
OBD-II or CAN pinout on DB9 connector.
See table for configurations.
MCP2551
U5
DB9 Plug
X1
F1
MBRA140
D1
PESD1CAN
D5
SJ4
SJ5
SJ6
Jumper Configurations
for DB9 pins (SJ4, SJ5, SJ6)
CAN
OBD-II
CAN-H
PIN 7
PIN 3
CAN-L
PIN 2
PIN 5
GND
PIN 3
PIN 2
MCP2515
U4
JP5
DNP
TP1
TP2
Q1
16MHz
www.skpang.co.uk
Released under the Creative Commons
Attribution Share-Alike 4.0 License
https://creativecommons.org/licenses/by-sa/4.0/
TITLE: SparkFun_CAN-Bus_Shield
Design by: Sukkin Pang
Revision By: Toni Klopfenstein
REV: v13
Date: 10/5/2015 12:45:20 PM
Sheet: 1/2
open hardware
sparkfun

Anexo 2

```
#include <SPI.h>
#include <Canbus.h>
#include <defaults.h>
#include <global.h>
#include <mcp2515.h>
#include <mcp2515_defs.h>
void setup()
{
   Serial.begin(115200); // Velocidad
}
    Serial.println("Inicializando comunicación CAN");
    delay(1000);
   if (Canbus.init(CANSPEED_500)) // Inicializamos velocidad de controlador.
    Serial.println("Conexión correcta de CAN");
   else
    Serial.println("No se pudo conectar el CAN");
}
Void loop()
{
  tCAN message;
  if (mcp2515_check_message() ) // Revisamos si hay mensajes disponibles
{
  if (mcp2515_get_message(&message)) //Devuelve el número de mensajes disponibles.
{
// if(message.id == 0x3B5 )  // Filtro de  mensajes.
//   {
  Serial.print("");              // Imprimimos en el monitor serial.
  Serial.print(times,6);
  Serial.print(" ");
  Serial.print(message.id, HEX);
  Serial.print("         ");
  Serial.print(" Rx");
  Serial.print("  ");
  Serial.print("d ");
  Serial.print(message.header.length, DEC);
  for (int i = 0; i < message.header.length; i++) // Se obtiene los datos de los mensajes
CAN.
 {
  Serial.print(message.data[i], HEX);
  Serial.print(" ");
}
  Serial.println("");}}}
```

Anexo 3

```
#include <SPI.h>
#include <Canbus.h>
#include <defaults.h>
#include <global.h>
#include <mcp2515.h>
#include <mcp2515_defs.h>
#include <SD.h>
File dataFile;                                          //Variable tipo archivo.

// Seleción de pin 9 en SD Card Shield
const int chipSelect = 9;
void setup()
{
   Serial.begin(115200); // Velocidad
   Serial.println("Inicializando tarjeta SD...");
   pinMode(chipSelect, OUTPUT);                         // ChipSelect como salida.
   if (!SD.begin(chipSelect)) //Verificamos si SD se ha inicializado.
{
    Serial.println("No hay una memoria disponible");
    return;                                             //Sale del programa.
}
    Serial.println("Tarjeta SD inicializada");
    Serial.println("Inicializando comunicacion CAN");
    delay(1000);
   if (Canbus.init(CANSPEED_500))                       // Inicializamos velocidad de controlador.
    Serial.println("Conexion correcta de CAN");
   else
    Serial.println("No se pudo conectar el CAN");
}
void loop()
{

  tCAN message;
  if (mcp2515_check_message() )                         // Revisamos si hay mensajes disponibles.
{
  if (mcp2515_get_message(&message)) //Devuelve el número de mensajes disponibles.
{
    String dataString = ""; }// Definimos una cadena.
  dataFile = SD.open("datas.TXT", FILE_WRITE);  // Abrimos un archive en la memoria SD.
  if ( time1==1)
{
   tiempo();
```

```
   time1=2;
}
  t_anterior = 0;
  t_anterior = millis();
  times = t_anterior * 0.001;//   times1 = t_anterior*1000;

// if(message.id == 0x3B5 )  // Filtro de  mensajes.
//   {
  Serial.print("");                                    // Imprimimos en el monitor serial.
  Serial.print(times,6);
  Serial.print(" ");
  Serial.print(message.id, HEX);
  Serial.print("         ");
  Serial.print(" Rx");
  Serial.print("  ");
  Serial.print("d ");
  Serial.print(message.header.length, DEC);
  dataFile.print(times,6);                          // Grabamos los mensajes en la memoria SD.
  dataFile.print(message.id, HEX);
  dataFile.print("         ");
  dataFile.print(" Rx");
  dataFile.print("    ");
  dataFile.print("d ");
  dataFile.print(message.header.length, DEC);
  dataFile.print(" ");
  for (int i = 0; i < message.header.length; i++)     // Se obtiene los datos de los mensajes.
 {
  Serial.print(message.data[i], HEX);
  Serial.print(" ");
  dataFile.print(message.data[i], HEX);
  dataFile.print(" ");
}
  dataFile.println();
  dataFile.close();                                    // Cerramos el archivo en la SD.
  Serial.println("");
}
}
}
```

Anexo 4

```
#include <Time.h>
#include <SPI.h>
#include <Canbus.h>
#include <defaults.h>
#include <global.h>
#include <mcp2515.h>
#include <mcp2515_defs.h>
#include <SD.h>
File dataFile;                                          //Variable tipo archivo.

// Seleción de pin 9 en SD Card Shield
const int chipSelect = 9;
void setup()
{
   Serial.begin(115200); // Velocidad
   Serial.println("Inicializando tarjeta SD...");
   pinMode(chipSelect, OUTPUT);                         // ChipSelect como salida.
   if (!SD.begin(chipSelect)) //Verificamos si SD se ha inicializado.
{
    Serial.println("No hay una memoria disponible");
    return;                                             //Sale del programa.
}
    Serial.println("Tarjeta SD inicializada");
    Serial.println("Inicializando comunicacion CAN");
    delay(1000);
   if (Canbus.init(CANSPEED_500))                 // Inicializamos velocidad de controlador.
    Serial.println("Conexion correcta de CAN");
   else
    Serial.println("No se pudo conectar el CAN");
}
void loop()
{

  tCAN message;
  if (mcp2515_check_message() )                 // Revisamos si hay mensajes disponibles.
{
  if (mcp2515_get_message(&message)) //Devuelve el número de mensajes disponibles.
{
    String dataString = ""; }// Definimos una cadena.
  dataFile = SD.open("datas.TXT", FILE_WRITE);  // Abrimos un archive en la memoria SD.
  if ( time1==1)
{
   tiempo();
```

```
   time1=2;
}
  t_anterior = 0;
  t_anterior = millis();
  times = t_anterior * 0.001;//   times1 = t_anterior*1000;

// if(message.id == 0x3B5 )  // Filtro de  mensajes.
//   {
  Serial.print("");                                          // Imprimimos en el monitor serial.
  Serial.print(times,6);
  Serial.print(" ");
  Serial.print(message.id, HEX);
  Serial.print("         ");
  Serial.print(" Rx");
  Serial.print("  ");
  Serial.print("d ");
  Serial.print(message.header.length, DEC);
 dataFile.print(times,6);  // Grabamos los mensajes en la memoria SD.
  dataFile.print(message.id, HEX);
  dataFile.print("         ");
  dataFile.print(" Rx");
  dataFile.print("    ");
  dataFile.print("d ");
  dataFile.print(message.header.length, DEC);
  dataFile.print(" ");
  for (int i = 0; i < message.header.length; i++)      // Se obtiene los datos de los mensajes.
 {
  Serial.print(message.data[i], HEX);
  Serial.print(" ");
  dataFile.print(message.data[i], HEX);
  dataFile.print(" ");
}
  dataFile.println();
  dataFile.close();                                     // Cerramos el archivo en la SD.
  Serial.println("");
}
}
}
void tiempo()    // Obtiene el tiempo con las funciones de la librería time y las almacena en
la memoria SD.
{
  time_t t=now();
  dataFile.print("Fecha de lectura:  ");
  dataFile.print(day(t));
```

```
  dataFile.print(+ "/");
  dataFile.print(month(t));
  dataFile.print(+ "/");
  dataFile.print(year(t));
  dataFile.print(" ");
  dataFile.print(hour(t));
  dataFile.print(+ ":");
  dataFile.print(minute(t));
  dataFile.print( ":");
  dataFile.print(second(t));
  dataFile.println();
  dataFile.close();
  Serial.print("Fecha de lectura:  ");
  Serial.print(day(t));
  Serial.print(+ "/");
  Serial.print(month(t));
  Serial.print(+ "/");
  Serial.print(year(t));
  Serial.print(" ");
  Serial.print(hour(t));
  Serial.print(+ ":");
  Serial.print(minute(t));
  Serial.print( ":");
  Serial.print(second(t));
  Serial.println();

}
```

Anexo 5

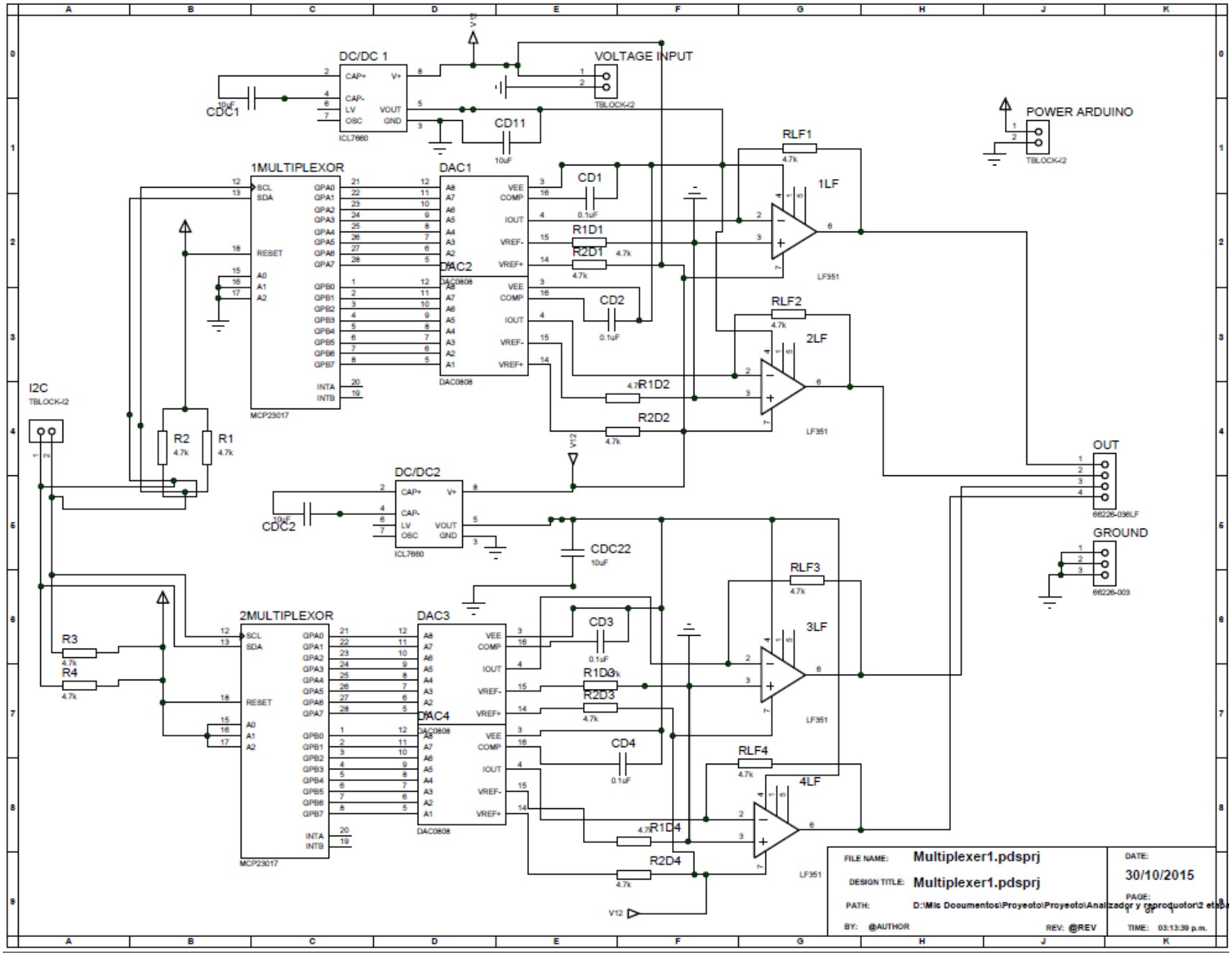
```
#include <stdlib.h>
char color=0;
int A=13;
  void setup() {
  Serial.begin(9600);
  delay(1000);
  Serial.println("@I20X4");                                    //inicializamos LCD.
  delayLCD();                                  //Función de retraso para el LCD.
  borrarLCD();                                                  // Borramos  LCD.
}
void loop() {
 borrarLCD();
 posicionLCD(2,2);                                 // Posiciona el cursor en la posición (2,2).
 enunciadoLCD("HOLA MUNDO ");          // Imprime en el lcd el enunciado HOLA MUNDO.
 numeroLCD(A);              //imprimimos en decimal el valor que tiene la variable "A".
  delayLCD();
}
void delayLCD(void)
{
  delay(100);
}
void borrarLCD(void)
{
  Serial.println("@B");
  delayLCD();
}
void posicionLCD(char renglon, char columna)  //Función para posicionar el cursor en el
LCD.
{char array[5];
  Serial.print("@P");
  itoa(renglon,array,10);
  Serial.print(array);
  Serial.write(',');
  itoa(columna,array,10);
  Serial.println(array);
  delayLCD();
}
void numeroLCD(int variable)          //Función para imprimir una variable de tipo entero.
{char array[7];
  itoa(variable,array,10);
 enunciadoLCD(array);
}
void enunciadoLCD(char *enunciado)                //Imprime  una cadena de caracteres.
```

```
{
  Serial.print("@E");
  Serial.println(enunciado);
  delayLCD();
}
void colorLCD(char numero_color)                    //Fija la luminosidad del LCD.
{char array[5];
  Serial.print("@L");
  itoa(numero_color,array,10);
  Serial.println(array);
  delayLCD();
}
```

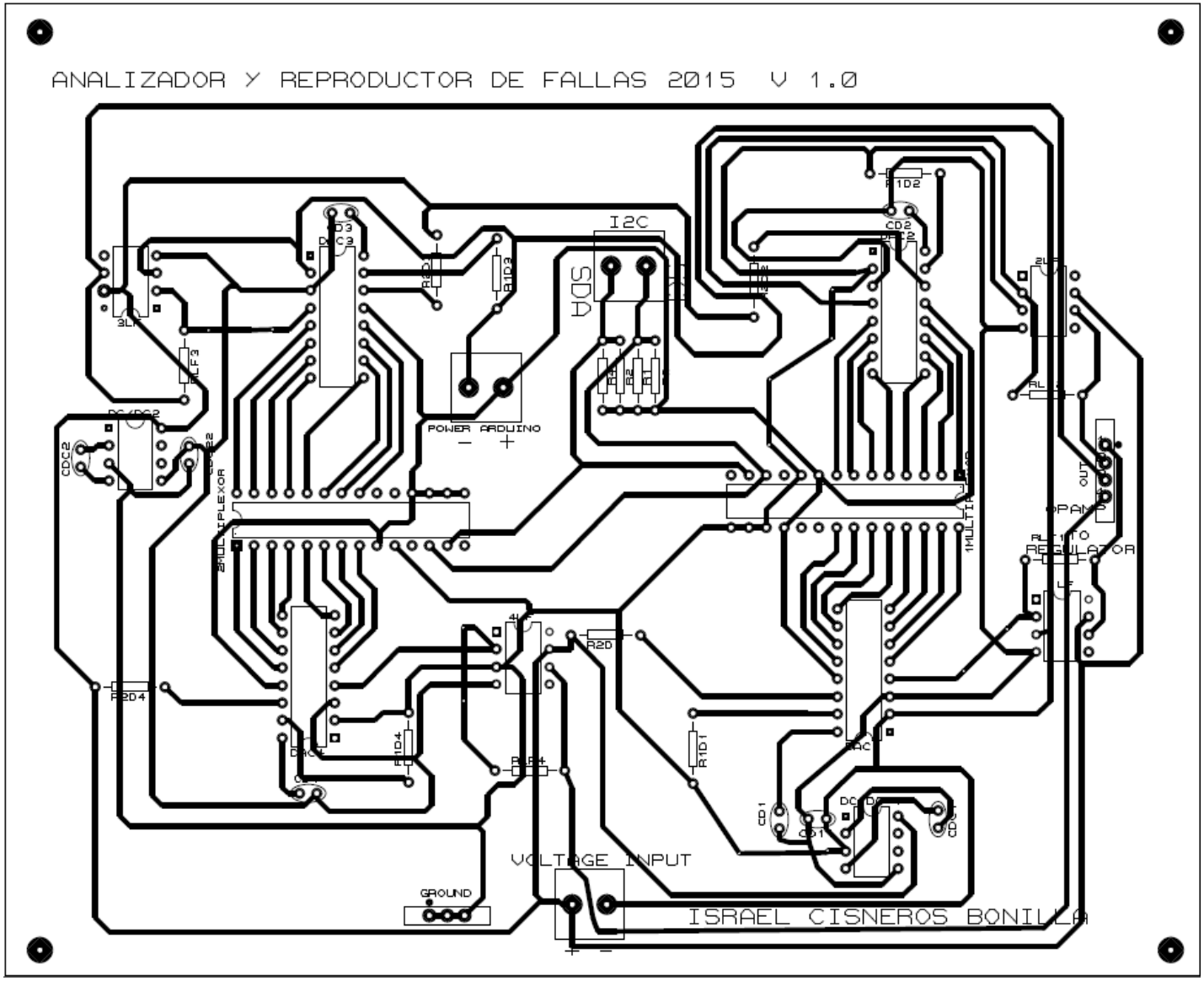
Anexo 6

```
#include "Wire.h"   //

void setup()

{
  Wire.begin(); // Despertamos el BUS I2C
  Wire.beginTransmission(0x20);   //  Dirección del dispositivo en el BUS.
  Wire.write(0x00);                          //IODIRA.
  Wire.write(0x00);                         // Todos el registro A como salida.
  Wire.endTransmission();                   //Finalizamos la comunicación.
  Wire.beginTransmission(0x20);    //Dirección del dispositivo en el BUS.
  Wire.write(0x01);                                                  // IODIRB.
  Wire.write(0x00);                         // Todos el registro B como salida.
  Wire.endTransmission();                   //Finalizamos la comunicación.
  Wire.beginTransmission(0x27);   //  Dirección del dispositivo en el BUS.
  Wire.write(0x00);                                                  //  IODIRA.
  Wire.write(0x00);                         // Todos el registro A como salida.
  Wire.endTransmission();                   //Finalizamos la comunicación.
  Wire.beginTransmission(0x27);     //Dirección del dispositivo en el BUS.
  Wire.write(0x01); //  IODIRB
  Wire.write(0x00); //   Todos el registro B como salida.
  Wire.endTransmission();  //Finalizamos la comunicación.
}
void loop()
{
  for (int i = 0; i < 255; i++)
  {
    Wire.beginTransmission(0x20);
    Wire.write(0x12);                              //  Dirección de puerto A.
    Wire.write(i);                             // Valor que enviara por el BUS.
    Wire.endTransmission();                          //Finaliza transmisión.
    Wire.beginTransmission(0x20);                 //  Dirección de puerto B.
    Wire.write(0x13); // GPIOB
    Wire.write(i);                             // Valor que enviara por el BUS.
    Wire.endTransmission();                    // Finalizamos comunicación.
   Wire.beginTransmission(0x27);
    Wire.write(0x12);
    Wire.write(i);
    Wire.endTransmission();
    Wire.beginTransmission(0x27);
    Wire.write(0x13); // GPIOB
    Wire.write(i);
    Wire.endTransmission();}}
```

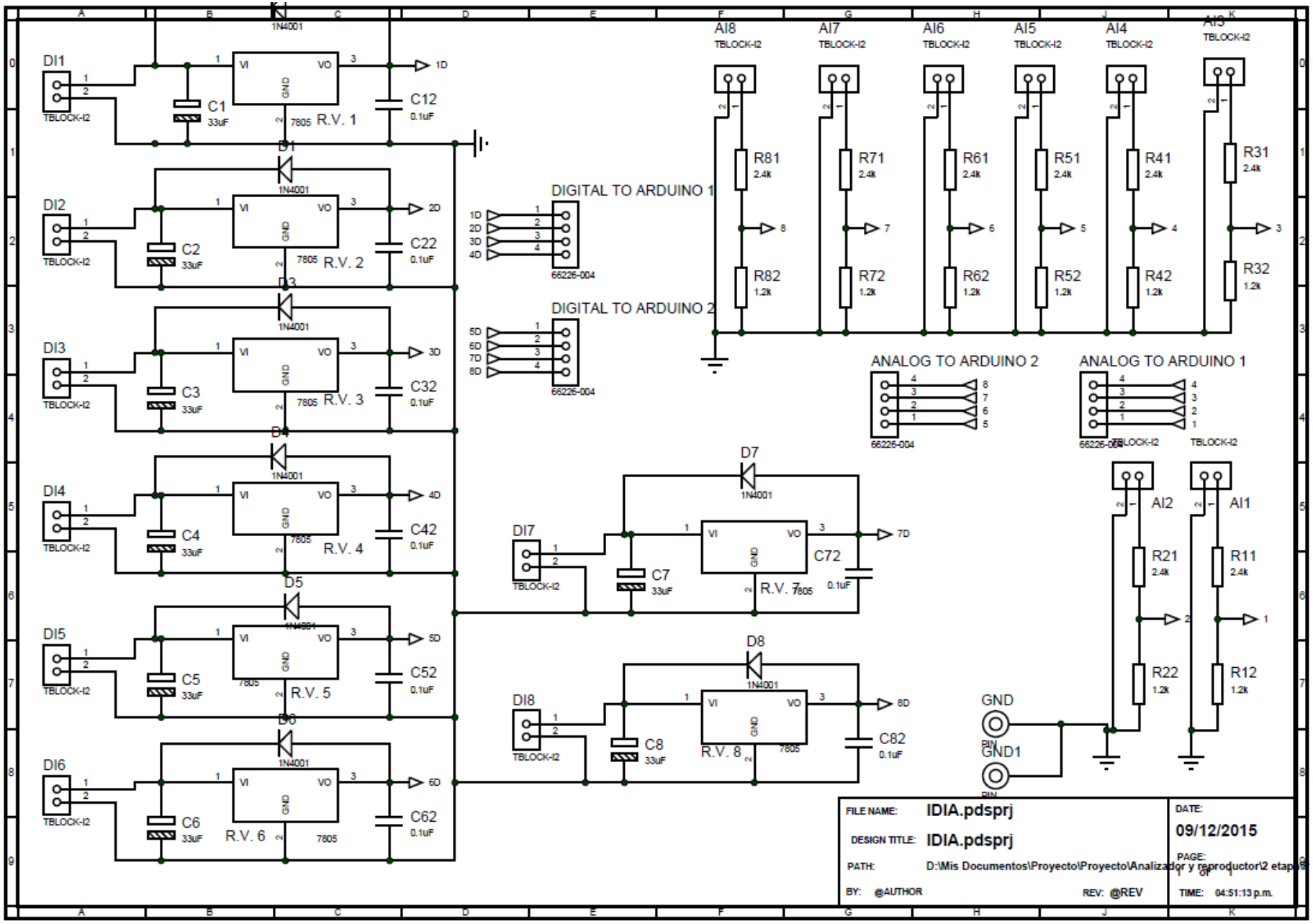
Anexo 7


DC/DC 1
VOLTAGE INPUT
POWER ARDUINO
CDC1
CD11
1MULTIPLEXOR
DAC1
DAC2
CD1
CD2
R1D1
R2D1
R1D2
R2D2
RLF1
RLF2
1LF
2LF
LF351
I2C
R1
R2
OUT
GROUND
DC/DC2
CDC2
CDC22
2MULTIPLEXOR
DAC3
DAC4
CD3
CD4
R1D3
R2D3
R1D4
R2D4
RLF3
RLF4
3LF
4LF
R3
R4
FILE NAME: Multiplexer1.pdsprj
DESIGN TITLE: Multiplexer1.pdsprj
PATH: D:\Mis Documentos\Proyecto\Proyecto\Analizador y reproductor\2 etapa
BY: @AUTHOR
REV: @REV
DATE: 30/10/2015
TIME: 03:13:39 p.m.

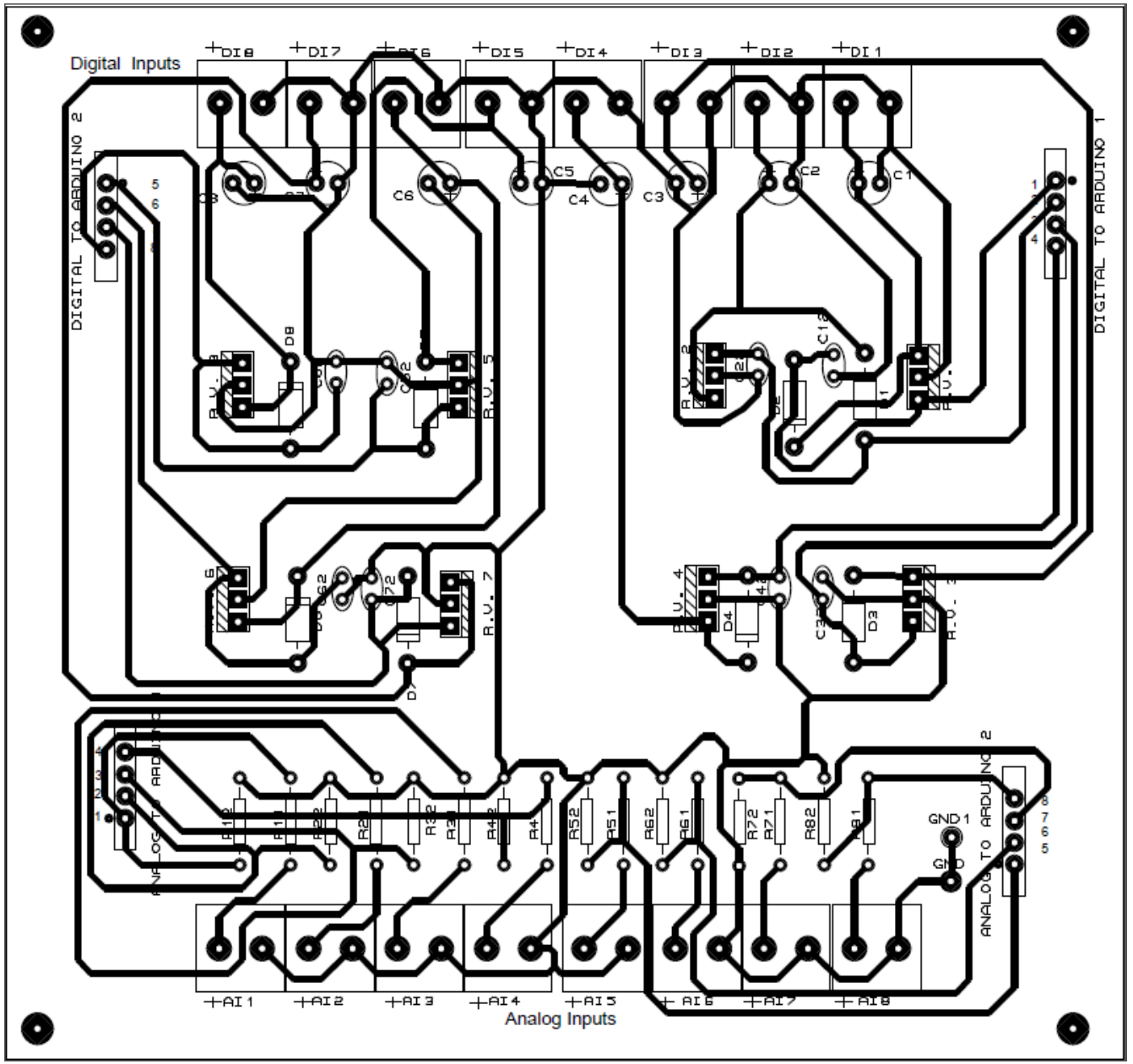
Anexo 8


ANALIZADOR Y REPRODUCTOR DE FALLAS 2015 V 1.0
I2C
SDA
POWER ARDUINO
VOLTAGE INPUT
GROUND
ISRAEL CISNEROS BONILLA

Anexo 9


DI1
DI2
DI3
DI4
DI5
DI6
DI7
DI8
TBLOCK-I2
1N4001
1N4884
VI
VO
GND
7805
C1
C2
C3
C4
C5
C6
C7
C8
33uF
C12
C22
C32
C42
C52
C62
C72
C82
0.1uF
R.V. 1
R.V. 2
R.V. 3
R.V. 4
R.V. 5
R.V. 6
R.V. 7
R.V. 8
D1
D2
D3
D4
D5
D6
D7
D8
1D
2D
3D
4D
5D
6D
7D
8D
DIGITAL TO ARDUINO 1
DIGITAL TO ARDUINO 2
66226-004
ANALOG TO ARDUINO 2
ANALOG TO ARDUINO 1
AI8
AI7
AI6
AI5
AI4
AI3
AI2
AI1
R81
R71
R61
R51
R41
R31
R21
R11
R82
R72
R62
R52
R42
R32
R22
R12
2.4k
1.2k
GND
GND1
PIN
FILE NAME: IDIA.pdsprj
DESIGN TITLE: IDIA.pdsprj
PATH: D:\Mis Documentos\Proyecto\Proyecto\Analizador y reproductor\2 etapa\
BY: @AUTHOR
REV: @REV
DATE: 09/12/2015
PAGE: 1 OF 1
TIME: 04:51:13 p.m.

Anexo 10


Digital Inputs
DI8
DI7
DI6
DI5
DI4
DI3
DI2
DI1
DIGITAL TO ARDUINO 2
DIGITAL TO ARDUINO 1
C1
C2
C3
C4
C5
C6
D1
D2
D3
D4
D5
D6
D7
D8
R.V. 1
R.V. 2
R.V. 3
R.V. 4
R.V. 5
R.V. 6
R.V. 7
R.V. 8
R11
R12
R21
R22
R31
R32
R41
R42
R51
R52
R61
R62
R71
R72
R81
R82
ANALOG TO ARDUINO 1
ANALOG TO ARDUINO 2
GND
AI1
AI2
AI3
AI4
AI5
AI6
AI7
AI8
Analog Inputs

Anexo 11

```
#define RELAY1  22    // Se definen los pines donde se conectaran los relevadores.
#define RELAY2  23
#define RELAY3  24
#define RELAY4  25
#define RELAY5  26
#define RELAY6  27
#define RELAY7  28
#define RELAY8  29
unsigned long previousMillis = 0;                  //Definimos el tiempo anterior.
unsigned long currentMillis;                       //Definimos el tiempo actual.
unsigned long previousMillis2 = 0;
unsigned long currentMillis2;
unsigned long previousMillis3 = 0;
unsigned long currentMillis3;
unsigned long previousMillis4 = 0;
unsigned long currentMillis4;
unsigned long previousMillis5 = 0;
unsigned long currentMillis5;
unsigned long previousMillis6 = 0;
unsigned long currentMillis6;
unsigned long previousMillis7 = 0;
unsigned long currentMillis7;
unsigned long previousMillis8 = 0;
unsigned long currentMillis8;
int ReleState = LOW;                          //Definimos el estado del relevador 1.
int ReleState2 = LOW;                         //Definimos el estado del relevador 2.
int ReleState3 = LOW;                          //Definimos el estado del relevador 3.
int ReleState4 = LOW;                         //Definimos el estado del relevador 4.
int ReleState5 = LOW;                         //Definimos el estado del relevador 5.
int ReleState6 = LOW;                         //Definimos el estado del relevador 6.
int ReleState7 = LOW;                         //Definimos el estado del relevador 7.
int ReleState8 = LOW;                         //Definimos el estado del relevador 8.
long intervalOn = 1000                  // Tiempo de encendido para el relevador 1.
long intervalOff = 1000;                   // Tiempo de apagado para el relevador 1.
long intervalOn2 = 1000                 // Tiempo de encendido para el relevador 2.
long intervalOff2 = 1000;                  // Tiempo de apagado para el relevador 2.
long intervalOn3 = 1000                 // Tiempo de encendido para el relevador 3.
long intervalOff3 = 1000;                 // Tiempo de apagado para el relevador 3.
long intervalOn4 = 1000                  // Tiempo de encendido para el relevador4.
long intervalOff4 = 1000;                  // Tiempo de apagado para el relevador 4.
long intervalOn5 = 1000                  // Tiempo de encendido para el relevador5.
long intervalOff5 = 1000;                  // Tiempo de apagado para el relevador 5.
```

```
long intervalOn6 = 1000                 // Tiempo de encendido para el relevador6.
long intervalOff6 = 1000;                  // Tiempo de apagado para el relevador 6.
long intervalOn7 = 1000                 // Tiempo de encendido para el relevador7.
long intervalOff7 = 1000;                  // Tiempo de apagado para el relevador 7.
long intervalOn8 = 1000                 // Tiempo de encendido para el relevador8.
long intervalOff8 = 1000;                  // Tiempo de apagado para el relevador 8.
void setup() {
  pinMode(RELAY1, OUTPUT);                    //Definimos los pines como salidas.
  pinMode(RELAY2, OUTPUT);
  pinMode(RELAY3, OUTPUT);
  pinMode(RELAY4, OUTPUT);
  pinMode(RELAY5, OUTPUT);
  pinMode(RELAY6, OUTPUT);
  pinMode(RELAY7, OUTPUT);
  pinMode(RELAY8, OUTPUT);
  digitalWrite(RELAY1, HIGH);                 //Definimos los pines con un valor alto.
  digitalWrite(RELAY2, HIGH);
  digitalWrite(RELAY3, HIGH);
  digitalWrite(RELAY4, HIGH);
  digitalWrite(RELAY5, HIGH);
  digitalWrite(RELAY6, HIGH);
  digitalWrite(RELAY7, HIGH);
  digitalWrite(RELAY8, HIGH);
}
void loop()
{
 unsigned long currentMillis = millis(); //Asignamos el valor que tiene el tiempo del Arduino
prendido.
  if (ReleState == LOW) {                    // Consulta el estado del relevador.
   if (currentMillis - previousMillis > intervalOff) // Si el tiempo actual es mayor al intervalo.
{
     previousMillis = currentMillis;
   ReleState = HIGH;                     // Desactiva relevador.
   }
  } else {
   if (currentMillis - previousMillis > intervalOn) Si el tiempo actual es mayor al intervalo de
prendido.
{
     previousMillis = currentMillis;
     ReleState = LOW;                     // Activa el relevador.
   }
  }
  digitalWrite(RELAY1, ReleState); // Escribe el valor digital en el pin del relevador.
  unsigned long currentMillis2 = millis();  //Relevador 2
```

```
if (ReleState2 == LOW) {
  if (currentMillis2 - previousMillis2 > intervalOff2) {
    previousMillis2 = currentMillis2;
    ReleState2 = HIGH;
  }
} else {
  if (currentMillis2 - previousMillis2 > intervalOn2) {
    previousMillis2 = currentMillis2;
    ReleState2 = LOW;
  }
}
digitalWrite(RELAY2, ReleState2);
 unsigned long currentMillis3 = millis();  //Relevador 3
if (ReleState3 == LOW) {
  if (currentMillis3 - previousMillis3 > intervalOff3) {
    previousMillis3 = currentMillis3;
    ReleState3 = HIGH;
  }
} else {
  if (currentMillis3 - previousMillis3 > intervalOn3) {
    previousMillis3 = currentMillis3;
    ReleState3 = LOW;
  }
}
digitalWrite(RELAY3, ReleState3);
   unsigned long currentMillis4 = millis();  //Relevador 4
if (ReleState4 == LOW) {
  if (currentMillis4 - previousMillis4 > intervalOff4) {
    previousMillis4 = currentMillis4;
    ReleState4 = HIGH;
  }
} else {
  if (currentMillis4 - previousMillis4 > intervalOn4) {
    previousMillis4 = currentMillis4;
    ReleState4 = LOW;
  }
}
digitalWrite(RELAY4, ReleState4);
     unsigned long currentMillis5 = millis();  //Relevador 5
if (ReleState5 == LOW) {
  if (currentMillis5 - previousMillis5 > intervalOff4) {
    previousMillis5 = currentMillis5;
    ReleState5 = HIGH;
  }
```

```
 } else {
   if (currentMillis5 - previousMillis5 > intervalOn5) {
     previousMillis5 = currentMillis5;
     ReleState5 = LOW;
   }
 }
 digitalWrite(RELAY5, ReleState5);
  unsigned long currentMillis6 = millis();  //Relevador 6
 if (ReleState6 == LOW) {
   if (currentMillis6 - previousMillis6 > intervalOff6) {
     previousMillis6 = currentMillis6;
     ReleState6 = HIGH;
   }
 } else {
   if (currentMillis6 - previousMillis6 > intervalOn6) {
     previousMillis6 = currentMillis6;
     ReleState6 = LOW;
   }  }
 digitalWrite(RELAY6, ReleState6);
unsigned long currentMillis7 = millis();  //Relevador 7
 if (ReleState7 == LOW) {
   if (currentMillis7 - previousMillis7 > intervalOff7) {
     previousMillis7 = currentMillis7;
     ReleState7 = HIGH;
   }
 } else {
   if (currentMillis7 - previousMillis7 > intervalOn7) {
     previousMillis7 = currentMillis7;
     ReleState7 = LOW;
   }}
 digitalWrite(RELAY7, ReleState7);
unsigned long currentMillis8 = millis();  //Relevador 8
 if (ReleState8 == LOW) {
   if (currentMillis8 - previousMillis8 > intervalOff8) {
     previousMillis8 = currentMillis8;
     ReleState8 = HIGH;
   }
 } else {
   if (currentMillis8 - previousMillis8 > intervalOn8) {
     previousMillis8 = currentMillis8;
     ReleState8 = LOW;
   }}
 digitalWrite(RELAY8,ReleState
```