Alejandro Ossio Díaz, A01209122
Gerardo Daniel Naranjo Gallegos, A01209499
Roberto Figueroa Saavedra, A01209689
03/12/2018

Proyecto PSoC

Contenido

1. Introducción

Los PSoC son microcontroladores programables desarrollados por Cypress Semiconductor. La principal ventaja que representa es la versatilidad de programación que incorpora el microcontrolador en un sistema configurable, dentro del mismo. Es decir, se puede configurar mediante el IDE cualquier configuración deseada, ya que cuenta con "componentes" internos configurables, desde *clock* hasta interrupciones, una inmensa variedad de configuraciones es posible. Entre todas las ventajas de las que dispone, se encuentra, por ejemplo, el poder definir cualquier puerto como se desea; esto quiere decir que otorga mayor libertad en el cableado externo, a diferencia de un Arduino que ya tiene sus pines fijos y limitados.

Adicionalmente, cuenta con la capacidad de incluir cualquier *shield* diseñado para Arduino, así como sensores y actuadores, incrementando las posibilidades.

Respecto a la programación, es muy versátil, ya que puede ser programado mediante código (lenguaje C) o mediante (*hardware*) conexiones físicas internas.

2. Material utilizado

El material utilizado para realizar esta práctica se enlista a continuación:

- Un PSoC 4 L-Series Pioneer Kit.
- Tres Arduino Uno.
- Una laptop con IDE Arduino y PSoC Creator.
- Cuatro cables (USB-Arduino).
- Cable CANBUS.
- Un *shield* joystick.
- Dos *shield* CANBUS.
- Un *shield power* driver.
- Una Fuente de alimentación de 12V.
- Cuatro intermitentes (dos bobinas y dos LED).
- Un faro principal.
- Un LED.
- Seis cables dupont macho-macho.
- Un *push-button*.
- Un protoboard.

3. Descripción de la práctica

A lo largo de esta práctica, se explicará cómo controlar cuatro intermitentes (dos focos de bobina y dos focos LED) y un faro principal, con función de luz alta y luz baja, en simulación del sistema de iluminación externa de un vehículo (excluyendo luces de freno y luces de posición). Actualmente, los vehículos cuentan con el protocolo de comunicación CANBUS para estos sistemas. Para lograrlo, se deberán conectar conforme a los esquemáticos, encontrados al final del documento en la sección de anexos.

La práctica consiste en cuatro etapas, la primera etapa referente al PSoC y una para cada Arduino utilizado.

En cuanto a la primera etapa, el PSoC será programado para identificar qué botones del sensor capacitivo, llamado CapSense, ha sido presionado. Con base en ello, se encenderán o apagarán las luces deseadas. Conforme a los botones oprimidos, se genera una cadena de caracteres que se enviará a la segunda etapa, al Arduino, para continuar el proceso.

La segunda y tercera etapa, que contemplan al segundo y tercer Arduino, llevarán el *shield* de Arduino CANBUS. A partir de ahora, a la segunda etapa se le llamará al Arduino que recibe mediante serial del primer Arduino y tendrá la función de transmitir mediante el CAN qué luces encender/apagar. Se le llamará tercera etapa al Arduino con la función de recibir el mensaje del segundo Arduino mediante CAN, para transmitir mediante serial al cuarto Arduino (o cuarta etapa).

Posteriormente a recibir mediante serial del tercer Arduino, el cuarto Arduino tendrá la función de encender/apagar las luces de la forma indicada desde el primer Arduino.

Como podemos observar, durante la segunda y tercera etapa, se implementará el protocolo de CANBUS para ponerlo en práctica. La primera y cuarta etapa funcionan a través de comunicación serial. Para información más detallada del protocolo CAN y las funcionalidades de cada etapa del proyecto, consultar las secciones posteriores.

4. Protocolo CANBUS:

En esta práctica se implementará comunicación utilizando Bus CAN. CAN proviene del inglés *Controller Area Network*, es un protocolo de comunicaciones.

Una característica interesante de este protocolo es que cada esclavo puede pasar a ser maestro.

Además, presenta diversas ventajas, como una alta inmunidad al ruido (perturbaciones/interferencias) y el economizar, ya que el cableado necesario es considerablemente menor, por ser una red multiplexada. Además, se presentan ventajas como la prioridad en mensajes, tiempos de latencia y una gran distinción de errores.

Físicamente, el CANBJS es un par de cable trenzado, como se puede observar en la siguiente imagen:

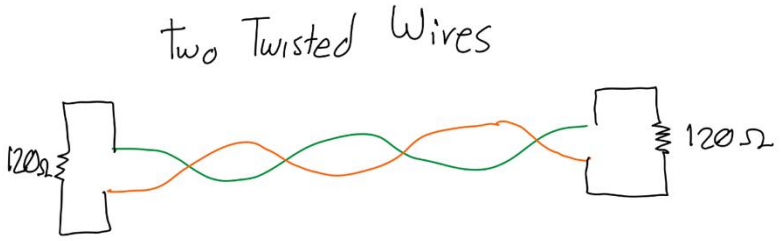


Figura 1. Tipo de cables del protocolo CAN.

Continuando con el apartado físico, podemos observar cómo se identifican los estados lógicos en la siguiente imagen.

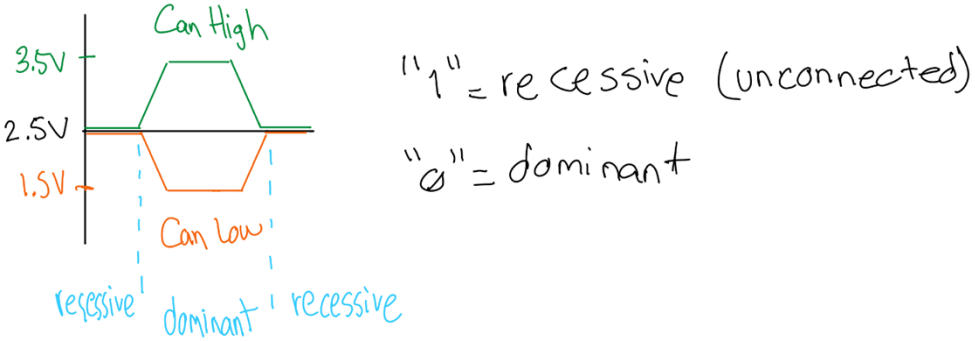


Figura 2. Estados lógicos del protocolo CAN.

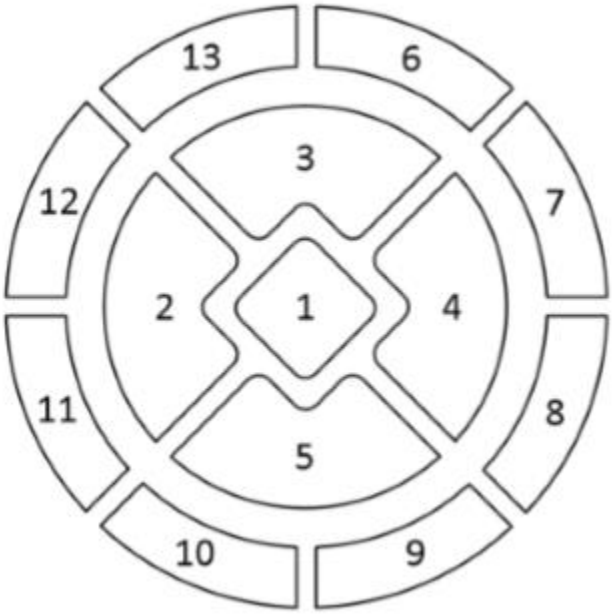
El protocolo CAN utiliza la siguiente estructura para su paquete de datos:

Figura 3. Paquete de datos de CAN.

5. PSoc

Este microcontrolador corresponde a la primera etapa del proyecto. Consiste de un PSoC, el cual cuenta con diversos puertos y sensores, como se puede apreciar en la Figura de la sección 13.1 y se programó con el código presentado en la sección 13.2.

El microcontrolador implementado en esta etapa presenta un sensor capacitivo, que cuenta con botones integrados, que será utilizado para encender o apagar las luces deseadas. Se muestra en la siguiente Figura 4, donde podemos identificar el número de pines del microcontrolador correspondientes a cada botón del sensor capacitivo.



Pad	Pin
1	P0[6]
2	P4[5]
3	P4[4]
4	P4[7]
5	P4[6]
6	P7[4]
7	P7[5]
8	P7[6]
9	P7[7]
10	P0[0]
11	P0[1]
12	P7[2]
13	P7[3]

Figura 4. CapSense.

Adicionalmente, se implementa una funcionalidad extra. Un botón específico activa la luz de cortesía; dicha función es un extra en la práctica. La luz de cortesía, o también conocida como función de *coming home* o *leaving home*, consiste en encender las luces del vehículo (cuando está apagado), con el propósito de iluminar por unos segundos el camino desde o hacia el vehículo.

Igualmente, se implementa la función de *dimmer*. Esto se logra mediante los botones periféricos 7 y 8. Con ellos, se logra aumentar o disminuir la cantidad de luz emitida por el faro principal; con el botón 8 se disminuye y con el botón 7 se aumenta. Esta función solamente es válida para la luz baja y luz alta, por lo tanto, no aplica en ninguna direccional ni al momento de encender la luz de cortesía.

La ocupación de este primer microcontrolador será reconocer qué botón ha sido oprimido, para mandar al Arduino de la etapa dos, mediante comunicación serial, las instrucciones necesarias. Previo a esto, se comprueban ciertas validaciones:

- Al activar las luces altas, se deben de desactivar las luces bajas. Esto debido a que la fuente utilizada no soportaría alimentar a ambas.
- Al activar una direccional, debe encender; si previamente estaba encendida la direccional opuesta, esta deberá apagarse.
- Al encender las intermitentes, tendrá prioridad sobre la función de direccionales. Si se encontraba encendida alguna direccional, esta deberá de volver a encenderse una vez apagadas las intermitentes. Además, si se apaga o se realiza un cambio de direccional mientras las intermitentes están encendidas, al apagarse las intermitentes se debe respetar este cambio o apagado.
- La luz de cortesía solamente debe encender el faro de la luz baja durante 10 segundos. Esta luz solamente encenderá si todo lo demás está previamente apagado (emulando que el carro estuviese apagado).
- El *dimmer* solamente funciona para el faro principal, en luces bajas y altas, pero no en las intermitentes ni en la función de luz de cortesía.

Todas las validaciones se llevan a cabo en el PSoC, mediante código.

En cuanto a la comunicación serial, primeramente, se genera una cadena de 8 caracteres; en donde se respeta el siguiente orden y donde un “0” representa apagado y un “1” representa encendido:

0) “#”: simboliza el inicio de la cadena y siempre es el mismo carácter.
1) “0” o “1”: representa a las intermitentes.
2) “0” o “1”: simboliza a la direccional derecha.
3) “0” o “1”: figura a la direccional izquierda.
4) “0” o “1”: utilizado para las luces altas.
5) “0” o “1”: requerido para las luces bajas.
6) “0” o “1”: implementado para la luz de cortesía.
7) Valor utilizado para el *dimmer*.

Posteriormente, con la información de qué luces encender, se envía la cadena de caracteres al Arduino de la segunda etapa mediante comunicación serial.

Para lograr la comunicación serial entre ambos, es necesario iniciarla y ajustar el *baud rate* (como se puede observar en el código) y conectar los puertos TX y RX (con los cables dupont).

Los puertos de transmisión y recepción se conectan de forma TX con RX y viceversa, es decir, un puerto transmisor se conecta al puerto receptor del otro. Finalmente, conectar la tierra entre ambos.

Cabe mencionar que las conexiones físicas de TX y RX se deben de desconectar cada que se desea reprogramar alguno de los dos microcontroladores. De no hacerlo, existe la posibilidad de dañarlos irremediablemente.

6. *Arduino CANBUS shield* transmisor

Este Arduino cumple la función de recibir, mediante comunicación serial, la cadena de caracteres del Arduino de la etapa 1, posteriormente, mandarla mediante CANBUS al Arduino de la etapa 3. Para lograr la comunicación serial, basta con conectar los puertos TX y RX del Arduino 1 y del Arduino 2.

En cuanto a la comunicación CANBUS, es requerido el *shield*. Para una mayor información, se muestra el esquemático en la sección 13.3.

El código del Arduino se puede encontrar en la sección 13.4. Básicamente, el Arduino debe de recibir la cadena de caracteres y reenviarla mediante CANBUS al Arduino tres, de la tercera etapa. Para esto se sigue un tipo de mensaje predefinido, en el cual lo primero que se envía el tipo de formato en que se va a enviar, en este caso nosotros estamos utilizando un 0x631 el cual corresponde a un formato hexadecimal, después lleva un *header* en el cual se especifica el número de bytes que se van a enviar, esto es equivalente a la longitud del mensaje, posteriormente se envía cada uno de los datos en el formato establecido y se finaliza el mensaje.

7. *Arduino CANBUS shield* receptor

Este Arduino recibe el mensaje mandado por el anterior mediante comunicación CAN, el código se puede encontrar en la sección 13.5, el cual se encarga de desempaquetar el mensaje CAN, leyendo el tipo de formato, después el número de bytes que se van a leer y hace un corrimiento para guardar cada uno en un arreglo, que posteriormente se convertirá para enviarlo mediante serial al próximo Arduino.

8. SparkFun power driver shield for Arduino

Este último Arduino, cuenta con el *shield* descrito en la Figura 6 y con el código de la sección 13.7. Su función es recibir la cadena de caracteres generada desde el PSoC, mediante comunicación serial, para poder identificar carácter por carácter y encender las luces correspondientes. Cabe mencionar que las validaciones se realizan desde el primer Arduino.

Esta etapa cuenta con una fuente de alimentación. En este caso particular, es una fuente de PC. Por lo tanto, se debe verificar que la fuente esté encendida para su correcto funcionamiento; es la alimentación de los faros e intermitentes.

Adicionalmente, este *shield* cuenta con un *switch* que se deberá encender.

La lógica del código es la inversa que la del Arduino en la primera etapa. Es decir, descompone la cadena de caracteres formada. Con base en cada carácter, se identifica cuáles luces deberán de ser encendidas. De esta forma, se manda la señal a través de los pines:

- D3: direccional izquierda trasera.
- D5: luz del faro baja.
- D6: direccional derecha delantera.
- D8: luz adicional agregada para identificar fallas.
- D9: direccional izquierda delantera.
- D10: direccional trasera derecha.
- D11: luz del faro alta.

Para mayor información, se puede consultar el esquemático de la sección 13.6.

9. Pasos para llevar a cabo

El método de elaboración de esta práctica consiste, como se ha dicho previamente, en cuatro etapas.

La primera etapa por llevar a cabo es programar la programación de todos los Arduinos con su respectivo código, ubicados en la sección de Anexos. Igualmente, para el PSoC; el proyecto se encuentra en los documentos adicionales a este reporte.

Posteriormente, a cada Arduino se le debe montar su *shield* necesario.

Consecutivamente, se deben de conectar los puertos necesarios, por ejemplo, en los pines de TX, RX y GND (tierra) de los Arduinos que tendrán la conexión serial. Es decir, el primero con el segundo y el tercero con el cuarto.

Además, conectar los cables de comunicación CANBUS entre el segundo y tercer Arduino.

Finalmente, conectar los cables de alimentación de los cuatro Arduinos y encender el HUB de alimentación.

Particularmente para el cuarto Arduino, se cuenta con la fuente de voltaje que alimentará a los faros. Dicha fuente, deberá de ser encendida. Después, se deberá colocar en la posición de encendido el *switch* con el que cuenta el case de dicho Arduino.

10. Resultados obtenidos

Los resultados del proyecto funcional es el control deseado del faro, en altas y bajas, y los cuatro intermitentes, con la función de direccionales e intermitentes. Contemplando en todos los casos sus respectivas validaciones e, incluso, con la implementación de extras.

11. Principales desafíos

Como principales retos a lo largo de la práctica realizada, se puede mencionar a las diversas validaciones a considerar, desde el primer Arduino. Más específicamente, las validaciones referentes a las intermitentes y direccionales.

Adicionalmente, los extras añadidos a la práctica, los cuales son:

- Lograr que las intermitentes y direccionales enciendan al mismo tiempo: Esto, debido a que las intermitentes traseras eran utilizadas en forma de LED y las delanteras con faros convencionales, por lo tanto, las traseras encendían primero, debido a que los faros LED no requieren calentar filamento, a diferencia de los convencionales. La solución se logró mediante la implementación de la función *delay* en el código.
- La implementación de una luz de cortesía: representó desafío lograr que la luz se encienda en 10 segundos, debido a factores internos del Arduino. Se solucionó con la función *delay* en el código.
- Finalmente, se incluyó una luz que indicaba algún error presentado: tratando de simular un testigo de *check engine* del clúster de instrumentos. El reto fue diseñar el circuito externo, aunque sencillo, que contemplara el *push button* adicional.

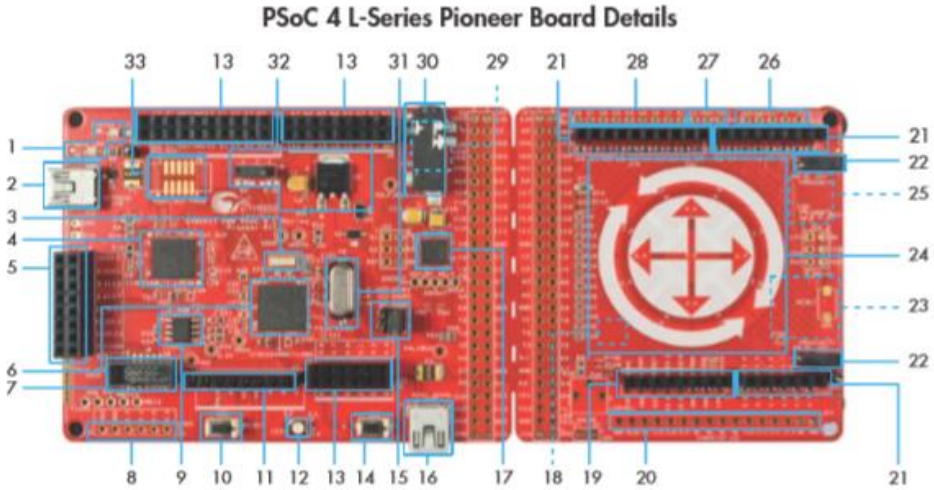
12. Referencias

García, A. (24 de junio de 2015). *Diseño de una red CAN bus con Arduino* . Obtenido de E.T.S. de Ingeniería Industrial, Informática y de Telecomunicación | Universidad Pública de Navarro: http://academica-e.unavarra.es/xmlui/bitstream/handle/2454/19115/TFG%20Diseño%20de%20una%20Red%20Can%20bus%20-%20Alejandro%20García%20Osés.pdf?sequence=1

GitHub. (18 de enero de 2017). *SparkFun_CAN-Bus_Arduino_Library* . Obtenido de GitHub: https://github.com/sparkfun/SparkFun_CAN-Bus_Arduino_Library

13. Anexos

13.1. Esquemático PSoC:



1. Power (LED2) and Status (LED1) LEDs
2. KitProg USB connector (J10)
3. 32.768 kHz crystal (Y1)
4. PSoC 5LP programmer and debugger (CY8C5868LTI-LP039, U4)
5. PSoC 5LP I/O header (J8)
6. PSoC 4200L (CY8C4248BZI-L489, U5)
7. PSoC 4200L program and debug header (J6)
8. Digilent® Pmod™ compatible I/O header (J5)*
9. Cypress F-RAM 1 Mb (FM24V10-G, U6)
10. PSoC 4200L reset button (SW1)
11. Arduino™ compatible main board power header (J1)
12. RGB LED (LED3)
13. Arduino compatible main board I/O headers (J2/J3/J4)
14. User button (SW2)
15. PSoC 4200L current measurement jumper (J14)
16. PSoC 4200L USB connector (J11)
17. Audio CODEC (WM8731CLSEFL, U3)
18. Cypress Energy Harvesting PMIC (S6AE101A, U12)*
19. Arduino compatible shield board power header (J17)
20. Character LCD header (J23)*
21. Arduino compatible shield board I/O headers (J18/J19/J20)
22. CapSense Proximity headers (J21, J22)
23. µSD card interface (J25) and Cypress Serial NOR Flash (S25FL512S, U13)*
24. CapSense® Gesture Pad
25. EZ-BLE™ PRoC™ Module (CYBLE-022001-00, U8)*
26. EZ-BLE Programming header (BLE PROG)*
27. Energy Harvesting PMIC header (EH)*
28. EZ-BLE I/O header (BLE)*
29. Audio jack selection (AHJ/OMTP) switch (SW4)*
30. Audio jack - 3.5 mm (J16)
31. 17.2032 MHz crystal (Y2)
32. System power selection jumper (J9) and LDO (3.3V, U2)
33. PSoC 5LP program and debug header (J7)*

*Footprint only, not populated on the board

Figura 5. Esquemático PSoC.

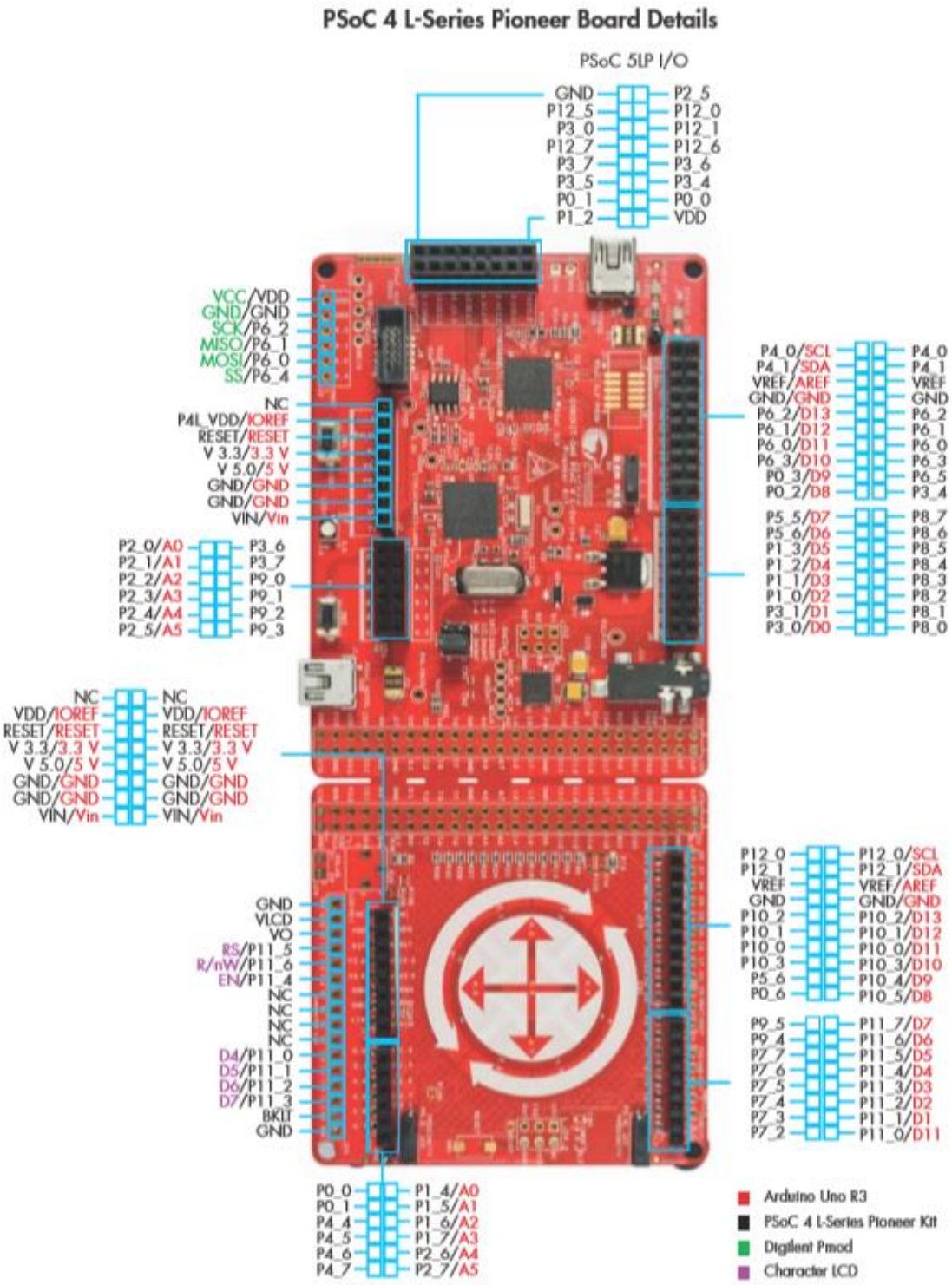


Figura 6. Esquemático PSoC.

13.2. Código para PSoC:

```
#include "main.h"
 //State flag
        int HAZARD = 0, TURNR = 0, TURNL = 0, HIGHB = 0, LOWB = 0,
COURTESY = 0, DIMMER = 0;
        //Previous state for turns
        int PrevTurnR = 0, PrevTurnL = 0, PrevHazard = 0;
        //Mechanical bounce flag
        int D4R = 0, D5R = 0, D2R = 0, D3R = 0, D6R = 0, D7R = 0, D8R;
        //String to send
        uint8 CAN[8] = {'0','0','0','0','0','0','0','0'};
        //Change of state
        int match = TRUE;
        //Variable for changes in turn while in hazard
        int HTurn = 0;
int main()
{
      #if(TUNER_ENABLE == ENABLED)
            /* Process tuner code if Tuner is enabled */
            CyGlobalIntEnable; /* Enable global interrupts. */

            /* Based on the type of tuner enabled, process appropriate
code */
            #if(TUNER_TYPE == TUNER_TYPE_BUTTONS)

                  CapSense_Buttons_TunerStart();

                  while(TRUE)
                  {
                        CapSense_Buttons_TunerComm();
                  }
            #endif
      #else
      /* Process example code if tuner is disabled */

            /* Variables to hold current and previous button status */
            uint32 buttonStatus = OFF, prevButtonStatus = OFF;

            /* Variable to store status of CSD0/CSD1 scan */
            uint32 scanStatus = 0;

        CyGlobalIntEnable; /* Enable global interrupts. */

          /* Initialize the CapSense buttons CSD block (CSD0) */
            CapSense_Buttons_Start();
            CapSense_Buttons_InitializeAllBaselines();

            /* Put the device to CPU sleep, WDT0 will wakeup the device
every 20 ms
                  This first sleep is executed to make sure all the
scanning happens
                  synchronously every 20 ms */
            CySysPmSleep();
```

```
            /* Trigger the initial scan of the CapSense buttons and
proximity sensor */
            CapSense_Buttons_ScanEnabledWidgets();

        Serial_Start();

          for(;;)
          {
                  /* Check if CSD0 (CapSense buttons) completed scanning
and the scan complete has not been processed */
                  if((CapSense_Buttons_IsBusy() != TRUE) && ((scanStatus
& CAPSENSE_BUTTON_SCAN_COMPLETE) != CAPSENSE_BUTTON_SCAN_COMPLETE))
                  {
                        /* Update CapSense buttons baselines */
                        CapSense_Buttons_UpdateEnabledBaselines();

                        /* If any widget is reported active, process the
button active status */
                        if(CapSense_Buttons_CheckIsAnyWidgetActive())
                        {
                              /* If more than one button is active,
suppress the unwanted sensors */
                              /* If only one sensor is active, the sensor
ON mask will be a power of 2 (only one bit set in the mask)
                                    The 'if' condition exploits a simple
property of 2^n number to check if the number is 2^n or not
                                    If it is 2^n, then only one sensor is
active. Otherwise more than one sensors are active

                                    Property of a 2^n number used below -
                                          2^n AND (Two's Complement(2^n))
= 2^n
                                    The above property is ONLY valid for
2^n numbers */

      if(IsNotPowerOfTwo(CapSense_Buttons_sensorOnMask[0]))
                              {
                                    /* Call the CapSense_ValidateButtons()
API to suppress unwanted buttons */
                                    buttonStatus =
CapSense_ValidateButtons(buttonStatus);
                              }
                              else
                              {
                                    /* If only one button is active, then
directly use the CapSense ON mask */
                                    buttonStatus =
CapSense_Buttons_sensorOnMask[0];
                              }
                        }
                        else
                        {
                              /* Clear the button status if no sensor is
active */
                              buttonStatus = CLEAR;
                        }
```

```
                        /* Set the CapSense scan complete flag for CSD0 */
                        scanStatus |= CAPSENSE_BUTTON_SCAN_COMPLETE;

                        /* If any button is active, then process LED
control activities */
                        if(buttonStatus != OFF)
                        {
                              /* Detect the rising edge on buttonStatus */
                              switch(RISING_EDGE_DET(buttonStatus,
prevButtonStatus))
                              {
                                    /* Left button activation, change LED
color from R > G > B > R */
                                    case (LEFT_BUTTON_MASK):
                               D6R = 1;
                                    break;

                                    /* Right button activation, change LED
color from R > B > G > R */
                                    case (RIGHT_BUTTON_MASK):
                                        D3R = 1;
                                    break;

                                    /* Up button activation, change LED
brightness from min to max */
                                    case (UP_BUTTON_MASK):
                                        D4R = 1;
                                    break;

                                    /* Down button activation, change LED
brightness from max to min */
                                    case (DOWN_BUTTON_MASK):
                                        D5R = 1;
                                    break;

                                    /* Centre button activation, toggle
LED ON/OFF state */
                                    case (CENTRE_BUTTON_MASK):
                                        D2R = 1;
                                    break;

                              /* Outer top left button activation, toggle LED
ON/OFF state */
                                    case (TOP1_BUTTON_MASK):
                                        D7R = 1;
                                    break;

                              /* Outer top left button activation, toggle LED
ON/OFF state */
                                    case (TOPRIGHT_BUTTON_MASK):
                                        D8R = 1;
                                    break;

                              /* Outer top left button activation, toggle LED
ON/OFF state */
                                    case (BOTRIGHT_BUTTON_MASK):
                                        D8R = 2;
```

```
                                    break;

                                    default:
                                    break;
                                }
                            }
                        }

                        /* Check if both CSD0 and CSD1 have completed their
scan */
                        if(scanStatus == CAPSENSE_SCAN_COMPLETE)
                        {
                            /* Store the current button status to previous
status */
                            prevButtonStatus = buttonStatus;

                            /* Put the device to CPU sleep as PWMs are running
and we cannot enter DeepSleep */
                        /* WDT0 is configured in the Clocks tab of .cydwr file to
generate an interrupt every 20 ms */
                            CySysPmSleep();

                            /* Clear the scan status flag and trigger the next
scan */
                            scanStatus = CLEAR;
                            CapSense_Buttons_ScanEnabledWidgets();
                        }
                    if(HAZARD == 0 && TURNR == 0 && TURNL == 0 && HIGHB == 0 &&
LOWB == 0){
                        if(D7R == 1){
                            COURTESY = 1;
                            match = FALSE;
                            D7R = 0;
                         }
                    }

                    if(COURTESY == 0){
                        if(D4R == 1){
                            LOWB = 0;
                            HIGHB = !HIGHB;
                            D4R = 0;
                            match = FALSE;
                        }
                    if(D5R == 1){
                        HIGHB = 0;
                        LOWB = !LOWB;
                        D5R = 0;
                        match = FALSE;
                    }
                    if(D2R == 1){
                        TURNR = 0;
                        TURNL = 0;
                        HAZARD = !HAZARD;
                        D2R = 0;
                        PrevHazard = 1;
                        match = FALSE;
                    }
```

```
if(HAZARD == 0){
    if(PrevHazard == 1){
        TURNR = PrevTurnR;
        TURNL = PrevTurnL;
    }

    if(HTurn == 1){
        TURNR = PrevTurnR;
        TURNL = PrevTurnL;
    }
    if(D3R == 1){
      TURNL = 0;
      TURNR = !TURNR;
      D3R = 0;
      match = FALSE;
    }
    if(D6R == 1){
      TURNR = 0;
      TURNL = !TURNL;
      D6R = 0;
      match = FALSE;
    }

    PrevTurnR = TURNR;
    PrevTurnL = TURNL;
    PrevHazard = 0;
    HTurn = 0;
  }
  else if(HAZARD == 1){
    if(D3R == 1){
        PrevTurnL = 0;
        PrevTurnR = !PrevTurnR;
        D3R = 0;
        match = FALSE;
    }
    if(D6R == 1){
        PrevTurnR = 0;
        PrevTurnL = !PrevTurnL;
        D6R = 0;
        match = FALSE;
    }
    HTurn = 1;
}
}
if(D8R == 1){
    DIMMER = 1;
    D8R = 0;
    match = FALSE;
}
if(D8R == 2){
    DIMMER = 2;
    D8R = 0;
    match = FALSE;
}

MakeOutput();
```

```
                if(match == FALSE)
                    Serial_SpiUartPutArray(CAN,8);
            match = TRUE;
            COURTESY = 0;
            DIMMER = 0;




          }

      #endif
}
/*******************************************************************************
*******
* Function Name: CapSense_ValidateButtons
********************************************************************************
*******
* Summary:
*  Validates the active buttons and suppresses the unwanted button
triggers
*
* Parameters:
*  prevButtonStatus - Previous button status
*
* Return:
*  uint32 - validated button status
*
********************************************************************************
******/
uint32 CapSense_ValidateButtons(uint32 prevButtonStatus)
{
      /* For loop variables */
      uint32 indexCount, indexButton;

      /* Variable to store current button status */
      uint32 currButtonStatus = CapSense_Buttons_sensorOnMask[0];


      /* Loop through all buttons */
      for(indexButton = 0; indexButton < (CapSense_Buttons_LEFT__BTN +
CapSense_Buttons_TOTAL_BUTTONS_COUNT - 1); indexButton++)
      {
            /* Check if the current button is active */
            if((1<<indexButton) & currButtonStatus)
            {
                  /* If the current button is active, then compare
against only the buttons that are not yet validated in the loop */
                  for(indexCount = indexButton+1; indexCount <
(CapSense_Buttons_LEFT__BTN + CapSense_Buttons_TOTAL_BUTTONS_COUNT);
indexCount++)
                  {
                        /* Check if the button in comparison is active */
                        if((1<<indexCount) & currButtonStatus)
                        {
```

```
                              /* Priority is for the button that was
active in the previous scan - so suppress the newly active button  */
                              if(prevButtonStatus & (0x01<<indexCount))
                              {
                                    currButtonStatus &=
~(0x01<<indexButton);
                                    break;
                              }
                              else if(prevButtonStatus &
(0x01<<indexButton))
                              {
                                    currButtonStatus &=
~(0x01<<indexCount);
                              }
                        /* If the button in comparison is active and neither
of them were active in the previous scan,
                              then compare the signal and declare the button
with stronger signal as active and suppress the other */
                              else
if(CapSense_Buttons_GetDiffCountData(indexButton) <
CapSense_Buttons_GetDiffCountData(indexCount))
                              {
                                    /* If current button has a weaker
signal, suppress the current button and break out of the comparison loop
*/
                                    currButtonStatus &=
~(0x01<<indexButton);
                                    break;
                              }
                              else
                              {
                                    /* If current button has a stronger
signal, suppress the button in comparison and move to next button for
comparison */
                                    currButtonStatus &=
~(0x01<<indexCount);
                              }
                        }
                  }
            }
      }

      /* Return the updated button mask */
      return currButtonStatus;
}

//Method to convert int to char:
char ConvertIntToChar(int entero){
  switch(entero)
  {
    case 2:
      return '2';
      break;
    case 1:
      return '1';
      break;
    case 0:
```

```
      return '0';
      break;
    default:
      return '?';
      break;
  }
}

//Method to build the matrix to send
void MakeOutput()
{
  CAN[0] = '#';
  CAN[1] = ConvertIntToChar(HAZARD);
  CAN[2] = ConvertIntToChar(TURNR);
  CAN[3] = ConvertIntToChar(TURNL);
  CAN[4] = ConvertIntToChar(HIGHB);
  CAN[5] = ConvertIntToChar(LOWB);
  CAN[6] = ConvertIntToChar(COURTESY);
  CAN[7] = ConvertIntToChar(DIMMER);
}

/* [] END OF FILE */
```

13.3. Esquemático Arduino CANBUS *shield*:

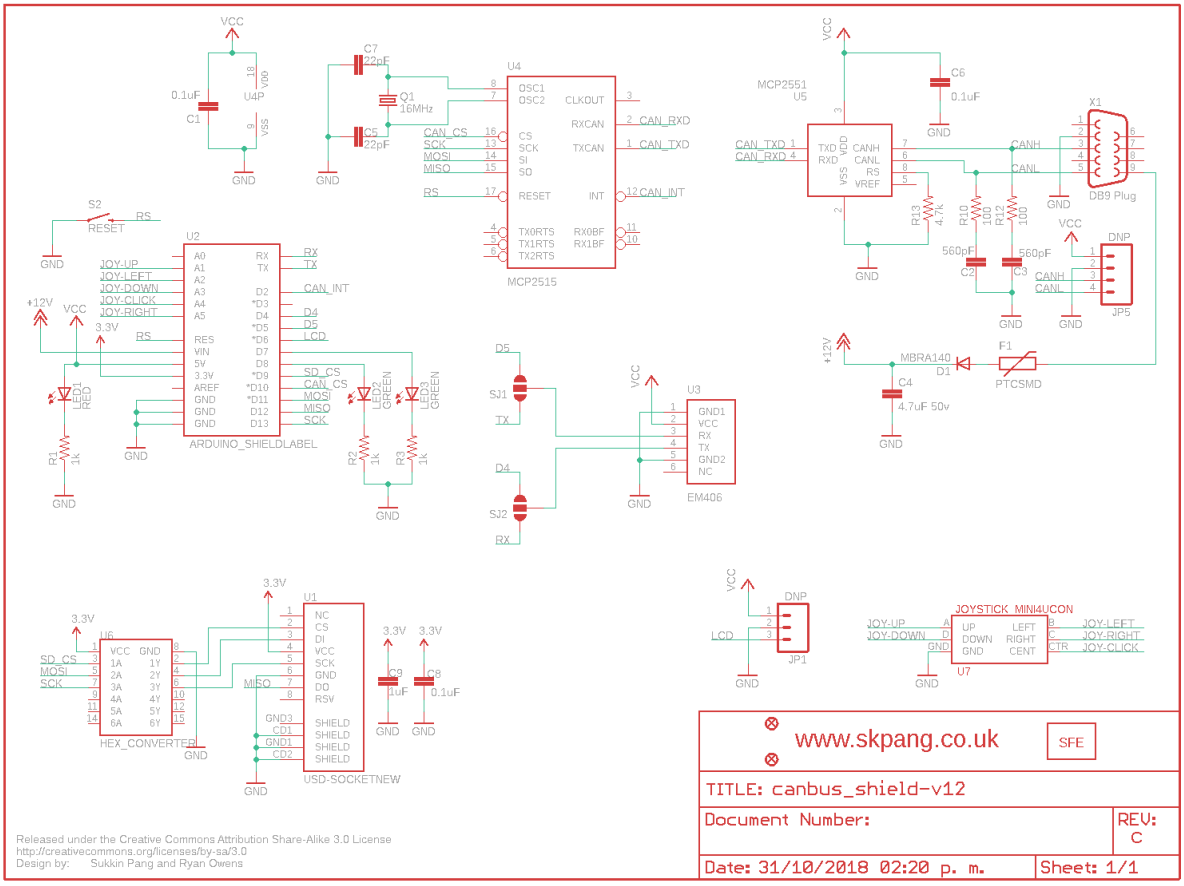


Figura 7. Esquemático CANBUS *Shield*.

13.4. Código para Arduino con SparkFun *CANBUS shield transmisor*:

```
#define BAUDRATE  9600

#include <Canbus.h>
#include <defaults.h>
#include <global.h>
#include <mcp2515.h>
#include <mcp2515_defs.h>

void setup() {
  Serial.begin(BAUDRATE);
  Canbus.init(CANSPEED_500);
}

//Variables;
char InputChar[8]    = {'x','x','x','x','x','x','x','x'};
bool Can             = false;

//Método de comunicación:
void Communication()
```

```
{
  if(Serial.available() > 0)
  {
    Serial.readBytes(InputChar, 8);
    Can = true;
  }
  else
    Can = false;
}

void Debug()
{
  Serial.print("Cadena = ");
  Serial.println(InputChar);
}

int ConvertionCharToHex(char toConvert)
{
  switch(toConvert)
  {
    case '0':
      return 0x30;
      break;
    case '1':
      return 0x31;
      break;
    case '2':
      return 0x32;
      break;
    case '#':
      return 0x23;
      break;
    case '&':
      return 0x26;
      break;
  }
}

void CAN()
{
  tCAN message;
  message.id = 0x631;         //formatted in HEX
  message.header.rtr = 0;
  message.header.length = 7;  //formatted in DEC
  message.data[0] = ConvertionCharToHex(InputChar[0]);
  message.data[1] = ConvertionCharToHex(InputChar[1]);
  message.data[2] = ConvertionCharToHex(InputChar[2]);
  message.data[3] = ConvertionCharToHex(InputChar[3]);
  message.data[4] = ConvertionCharToHex(InputChar[4]);
  message.data[5] = ConvertionCharToHex(InputChar[5]);
  message.data[6] = ConvertionCharToHex(InputChar[6]);
  mcp2515_bit_modify(CANCTRL, (1<<REQOP2)|(1<<REQOP1)|(1<<REQOP0), 0);
  mcp2515_send_message(&message);
}

void loop()
{
```

```
  Communication();
  if(Can){
    Debug();
    CAN();
  }
}
```

13.5. Código para Arduino con SparkFun *CANBus shield receptor*:

```
#define BAUDRATE  9600

#include <Canbus.h>
#include <defaults.h>
#include <global.h>
#include <mcp2515.h>
#include <mcp2515_defs.h>

char Output[8] = {'x','x','x','x','x','x','x','x'};

void setup() {
  Serial.begin(BAUDRATE);
  Canbus.init(CANSPEED_500);
}

char ConvertionHexToChar(int toConvert)
{
  switch(toConvert)
  {
    case 0x30:
      return '0';
      break;
    case 0x31:
      return '1';
      break;
    case 0x32:
      return '2';
    case 0x26:
      return '&';
      break;
    case 0x23:
      return '#';
      break;
  }
}

void ReadCAN()
{
  tCAN message;
  if (mcp2515_check_message())                                    //Check for
available message (recibing from CANBUS Joystick)
  {
    if (mcp2515_get_message(&message))                            //Read the
message
    {
      for(int i=0;i<message.header.length;i++)
      {
          Output[i] = ConvertionHexToChar(message.data[i]);   //Saving
the convertion from HEX to Char in Array "Output" the message
      }
    }
    Serial.write(Output,8);                                       //Write in
Serial the message in Char
  }
```

```
}

void loop()
{
  ReadCAN();
}
```

13.6. Esquemático SparkFun *power driver shield*:

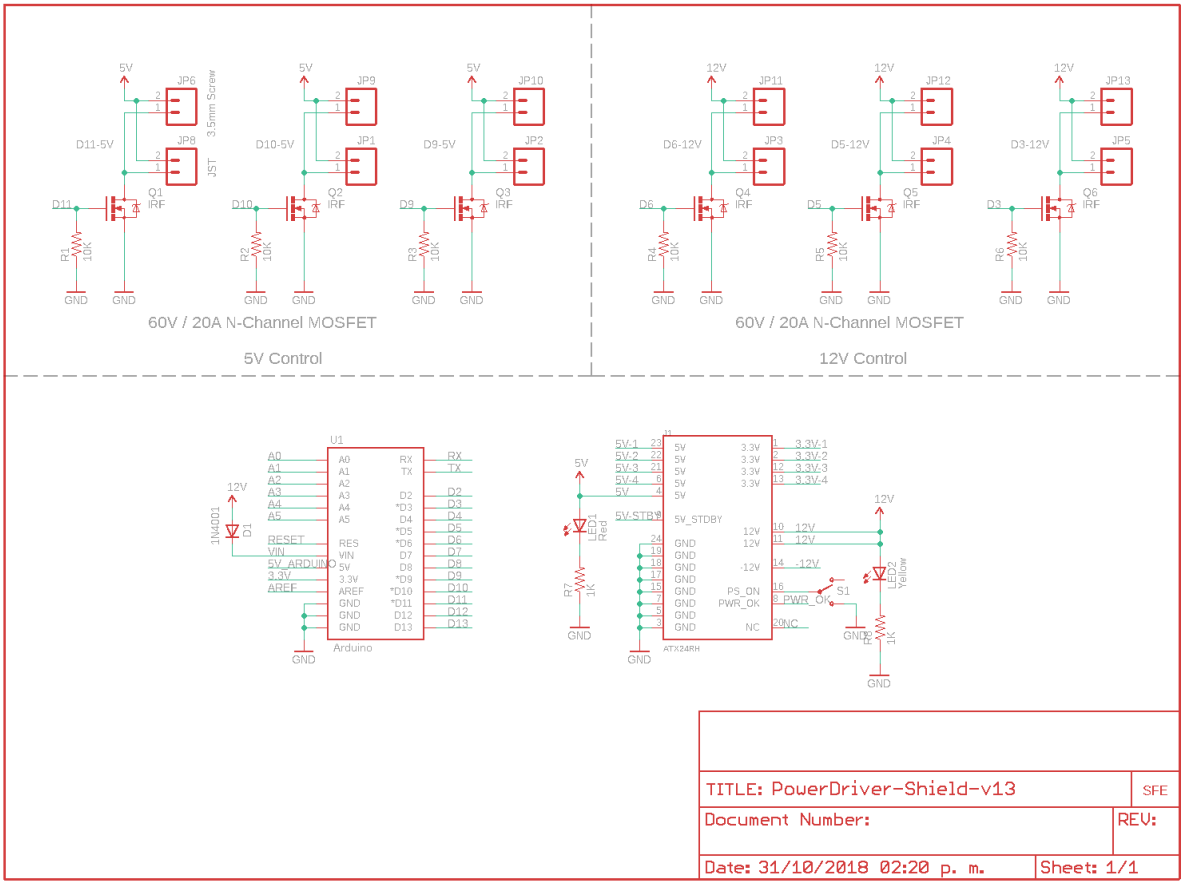


Figura 8. Esquemático *power driver shield*.

13.7. Código para Arduino con SparkFun *power driver shield*:

```
#define pin_FrontRight 6
#define pin_FrontLeft 9
#define pin_BackRight 10
#define pin_BackLeft 3
#define pin_HighBeam 11
#define pin_LowBeam 5
#define pin_Warning 8
#define BAUDRATE  9600

char InputChar[8] = {'x','x','x','x','x','x','x','x'};
int Dimmer = 128;

void setup() {
  Serial.begin(BAUDRATE);
  pinMode(pin_FrontRight, OUTPUT);
  pinMode(pin_FrontLeft, OUTPUT);
  pinMode(pin_BackRight, OUTPUT);
  pinMode(pin_BackLeft, OUTPUT);
  //pinMode(pin_HighBeam, OUTPUT);
```

```
  //pinMode(pin_LowBeam, OUTPUT);
  pinMode(pin_Warning, OUTPUT);
}

void Debug()
{
  Serial.print("Cadena = ");
  Serial.println(InputChar);
}

void Communication()
{
  if(Serial.available() > 0)
  {
    Serial.readBytes(InputChar, 8);
    Debug();
  }
}

void HAZARD()
{
  digitalWrite(pin_FrontRight, HIGH);
  delay(25);
  digitalWrite(pin_BackRight, HIGH);
  digitalWrite(pin_FrontLeft, HIGH);
  delay(25);
  digitalWrite(pin_BackLeft, HIGH);
  delay(500);
  digitalWrite(pin_FrontRight, LOW);
  delay(25);
  digitalWrite(pin_BackRight, LOW);
  digitalWrite(pin_FrontLeft, LOW);
  delay(25);
  digitalWrite(pin_BackLeft, LOW);
  delay(500);
}

void NOHAZARD(){
  digitalWrite(pin_FrontRight, LOW);
  digitalWrite(pin_BackRight, LOW);
  digitalWrite(pin_FrontLeft, LOW);
  digitalWrite(pin_BackLeft, LOW);
  }

void TURNLEFT()
{
  digitalWrite(pin_FrontLeft, HIGH);
  delay(25);
  digitalWrite(pin_BackLeft, HIGH);
  delay(500);
  digitalWrite(pin_FrontLeft, LOW);
  delay(25);
  digitalWrite(pin_BackLeft, LOW);
  delay(500);
}
void NOTURNLEFT(){
  digitalWrite(pin_FrontLeft, LOW);
```

```
  digitalWrite(pin_BackLeft, LOW);
  }

void TURNRIGHT()
{
  digitalWrite(pin_FrontRight, HIGH);
  delay(25);
  digitalWrite(pin_BackRight, HIGH);
  delay(500);
  digitalWrite(pin_FrontRight, LOW);
  delay(25);
  digitalWrite(pin_BackRight, LOW);
  delay(500);
}
void NOTURNRIGHT(){
  digitalWrite(pin_FrontRight, LOW);
  digitalWrite(pin_BackRight, LOW);

  }

void HIGHBEAM()
{
  analogWrite(pin_HighBeam, Dimmer);
}

void NOHIGHBEAM(){
  analogWrite(pin_HighBeam, 0);
  }

void LOWBEAM()
{
  analogWrite(pin_LowBeam, Dimmer);
}

void NOLOWBEAM(){
  analogWrite(pin_LowBeam, 0);
  }

void COURTESY(){
  analogWrite(pin_LowBeam, 255);
  delay(10000);
  analogWrite(pin_LowBeam, 0);
  }

void Validations()
  {
    if((InputChar[1] == '0' || InputChar[1] == '1') && InputChar[2] ==
'0' && InputChar[3] == '0' && InputChar[4] == '0' && InputChar[5] == '0'
){
      digitalWrite(pin_FrontRight, LOW);
      digitalWrite(pin_BackRight, LOW);
      digitalWrite(pin_FrontLeft, LOW);
      digitalWrite(pin_BackLeft, LOW);
      digitalWrite(pin_LowBeam, LOW);
      digitalWrite(pin_HighBeam, LOW);
    }
    if(InputChar[1] == '1'){
```

```
        HAZARD();
      }
    else if (InputChar[1] == '0'){
      NOHAZARD();
      }
    if(InputChar[2] == '1'){
        TURNRIGHT();
      }
    else if (InputChar[2] == '0'){
      NOTURNRIGHT();
      }
    if(InputChar[3] == '1'){
        TURNLEFT();
      }
    else if (InputChar[3] == '0'){
      NOTURNLEFT();
      }
    if(InputChar[4] == '1'){
        HIGHBEAM();
      }
    else if (InputChar[4] == '0'){
      NOHIGHBEAM();
      }
    if(InputChar[5] == '1'){
        LOWBEAM();
      }
    else if (InputChar[5] == '0'){
      NOLOWBEAM();
      }
    if(InputChar[6] == '1'){
      COURTESY();
      }
    if(InputChar[7] == '1'){
      Dimmer = Dimmer + 10;
      if(Dimmer >= 255){
        Dimmer = 255;
      }
    }
    if(InputChar[7] == '2'){
        Dimmer = Dimmer - 10;
      if(Dimmer <= 64){
        Dimmer = 64;
      }
    }
  }

void loop() {
  Communication();
  if(InputChar[0] == '#'){
    Validations();
    InputChar[6] = '0';
    InputChar[7] = '0';
    digitalWrite(pin_Warning,LOW);
  }
  else
    digitalWrite(pin_Warning,HIGH);
}
```