Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño de un dispositivo HID basado en microcontrolador

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Torrella Villaplana, Sergio

Tutor: Rodas Jordá, Ángel

2014-2015

Resumen

En el presente trabajo se aborda el proceso completo de creación de un dispositivo HID (Human Interface Device) basado en un microcontrolador. En concreto, se utiliza el microcontrolador ATMega2560 que incorpora la placa Arduino junto a ciertos sensores y actuadores para desarrollar la parte *hardware* y el entorno de desarrollo de Arduino para la parte *software*. El *firmware* es el encargado de proporcionar la interfaz HID para que el *host* reconozca el dispositivo, el cual es programado en el microcontrolador ATMega16U2 que hace de puente entre USB y la conexión RS232 al microcontrolador principal de la placa. A lo largo del proyecto se abordan varios ejemplos de dispositivos donde se utilizan *firmwares* desarrollados por terceros y finalmente se desarrolla uno propio con la ayuda de la librería LUFA.

Palabras clave: Arduino, HID, microcontrolador.

Abstract

The present project tackles the whole creation process of a Human Interface Device based on a microcontroller. Specifically, the Arduino built-in microcontroller with some sensors and actuators are used to develop the hardware and the Arduino development environment to develop the software. The firmware has the responsibility of providing the HID interface in order to give to the host the ability to understand the developed device. That firmware is programmed on the microcontroller ATMega16U2, which acts as a bridge between the USB connection and the main microcontroller of the board. Throughout the project some examples of different devices are made using third-party firmwares and then a custom firmware is developed with the help of LUFA's library.

Keywords: Arduino, HID, microcontroller.

Tabla de contenidos

1. Introducción 7
1.1 Motivación 7
1.2 Objetivos y alcance del proyecto 7
2. Dispositivos HID 7
2.1 Teclado 8
2.2 Ratón 8
2.3 *Joystick* 8
3. Herramientas empleadas 8
3.1 *Hardware* 9
3.1.1 Arduino Mega 2560 (R3) 9
3.1.2 Pulsador 11
3.1.3 Potenciómetro 12
3.1.4 *Joystick* 12
3.1.5 *Display* 7 segmentos 13
3.1.6 Sensor de ultrasonidos HC-SR04 13
3.1.7 *Display* LCD 16x2 14
3.1.8 Teclado 4x4 15
3.2 *Software* 16
3.2.1 Arduino IDE 16
3.2.2 Visual Studio 2013 (CE) + Visual Micro 17
3.2.3 Fritzing 17
3.2.4 Flip 18
4. Dispositivos desarrollados 18
4.1 Teclado con teclas personalizables 19
4.2 Teclado musical MIDI 24
4.3 Ratón de ultrasonidos 27
4.4 Mando para juegos 31
5. Creación de un *firmware* 33
5.1 Protocolo de mensajes 33
5.2 Entorno de desarrollo 34
5.3 Desarrollo del *firmware* 34
5.3.1 Descriptores HID 34

5.3.2 Envío de mensajes al *host* ... 35

5.3.3 Implementación del protocolo de mensajes ... 36

5.3.4 Adaptación del protocolo al *software* del mando para juegos ... 38

6. Conclusiones ... 38

7. Referencias ... 39

1. Introducción

1.1 Motivación

Los dispositivos HID (*Human Interface Device*) permiten la interacción de las personas con los computadores. Muchos de estos dispositivos son limitados en cuanto al *hardware*, de modo que apenas tienen diferencias entre ellos. El hecho de poder diseñar un dispositivo HID propio nos da la posibilidad de adaptar los dispositivos a nuestras necesidades en vez de tener que adaptarnos nosotros a los dispositivos que hay en el mercado.

Se ha elegido la plataforma Arduino para el desarrollo de este proyecto dado que incorpora un primer microcontrolador principal para manejar el hardware y un segundo microcontrolador que hace de puente entre el primero y un puerto USB y que, además, puede ser programado. De este modo se simplifica bastante el desarrollo de la parte del *hardware*, aunque se podría haber programado todo directamente sobre un único microcontrolador que tuviese interfaz USB.

1.2 Objetivos y alcance del proyecto

El principal objetivo de este proyecto viene dado por el título del mismo: Diseñar un dispositivo HID basado en microcontrolador. Para ello se construirán varios ejemplos de dispositivos mediante el uso de un microcontrolador Arduino y distintos sensores conectados a éste.

Sabiendo los temas a tratar para lograr el objetivo principal podemos extraer una serie de objetivos secundarios:

- Investigar y comprender el funcionamiento básico de la parte del *firmware* (comunicación entre el *hardware* del periférico y el *software* del ordenador) de los dispositivos HID.
- Programación del sistema microcontrolador basado en Arduino.
- Creación un *firmware* para la comunicación entre Arduino y el ordenador, bien creándolo desde cero o bien adaptando uno ya existente a nuestras necesidades particulares.

2. Dispositivos HID

La palabra HID, en español dispositivo de interfaz humana, hace referencia a un tipo de interfaces de usuario para ordenadores que interactúan directamente entre el humano y un *host*, dando la posibilidad tanto de entrada como de salida de información.

En el protocolo HID existen 2 entidades: el *host* y el dispositivo. El dispositivo es el encargado de interactuar directamente con el humano mientras que el *host* se comunica con el dispositivo intercambiando datos. En este protocolo, los dispositivos definen sus paquetes de datos y luego presentan un descriptor HID al *host*. El descriptor HID consiste en un grupo de bytes que describen los paquetes de datos del dispositivo (cuantos paquetes soporta, el tamaño de los paquetes y el propósito de cada byte/bit del paquete).

Es un protocolo reconocido por la mayoría de sistemas operativos, por lo que no se necesita controladores propietarios si no que la conexión es del tipo *plug & play*.

A continuación se presenta una breve descripción de los dispositivos HID más conocidos:

2.1 Teclado

El teclado es un dispositivo de entrada que utiliza una disposición determinada de teclas para que actúen como pulsadores que envían información al *host*. El principal uso del teclado es la introducción de texto. La disposición de teclas más conocida y utilizada en España es QWERTY. Además del bloque de introducción de texto, el teclado suele contar con un bloque de funciones (F1-F12), un bloque numérico que facilita la introducción de cifras y algunos también cuentan con atajos implementados por el fabricante.

2.2 Ratón

El ratón es un dispositivo de entrada que se encarga de facilitar la interacción con el entorno gráfico del *host*. El más sencillo cuenta con únicamente un botón y la detección del desplazamiento en dos ejes. Adicionalmente, es común encontrarlos con un segundo botón y con una rueda de desplazamiento entre los dos botones, que a su vez también realiza la función de botón. También suelen contar con más botones como por ejemplo los de navegación atrás y adelante o de cambio de sensibilidad de lectura PPP (puntos por pulgada). Actualmente los hay mecánicos, ópticos y de láser, tanto cableado como inalámbricos.

2.3 *Joystick*

El *joystick* es un dispositivo de entrada cuya función principal es su uso en videojuegos y simuladores, o como sustituto del ratón en personas con discapacidad. Está formado por, al menos, una palanca de dos ejes y un botón, aunque es habitual encontrarlo con varios botones y más de una palanca. La disposición suele variar bastante entre los fabricantes aunque lo normal es que tengan forma de mando o forma de palanca pegada a una base. Se distingue entre *joysticks* digitales (utilizan dos interruptores en cada eje) y analógicos (utilizan un potenciómetro en cada eje).

3. Herramientas empleadas

En este apartado se describen las herramientas que se utilizan a lo largo del trabajo. En primer lugar los dispositivos *hardware* con los que se han diseñado los periféricos y posteriormente el *software* empleado en el proceso de construcción de los mismos.

3.1 *Hardware*

3.1.1 Arduino Mega 2560 (R3)

Arduino es una plataforma de *hardware* libre basada en una placa con microcontrolador (en nuestro caso un ATmega2560) y una serie pines de entrada/salida (en nuestro caso 54). El Arduino Mega 2560 (Revisión 3) se comunica con el ordenador a través de un cable USB, haciendo de puente entre ambos un microcontrolador ATMega16U2.

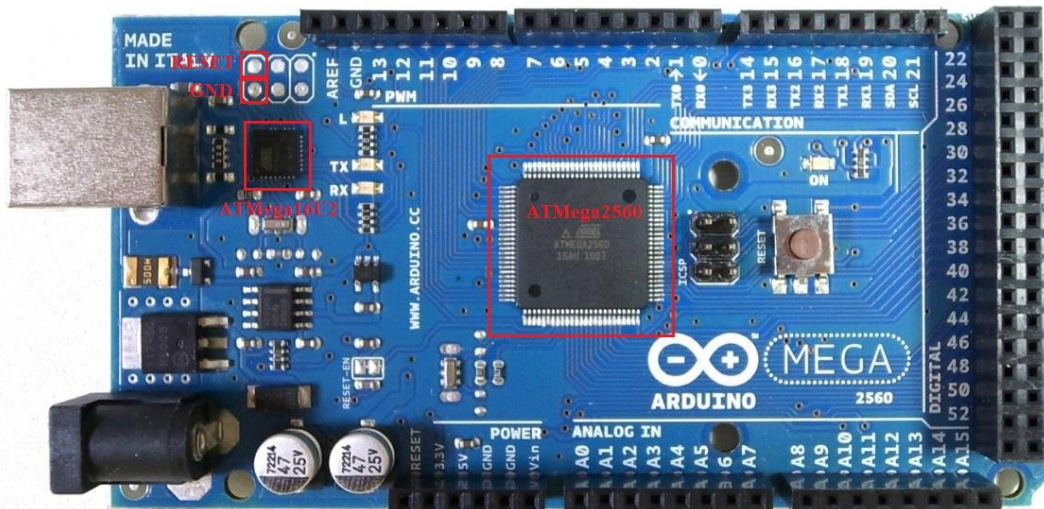

Ilustración 1. Placa de Arduino Mega 2560

A continuación se enumeran las características principales de la placa:

Alimentación

La placa tiene la posibilidad de alimentarse a través de la conexión USB y también a través de una fuente de alimentación externa. Para utilizar una fuente externa, se puede conectar a través del conector hembra con centro positivo que tiene integrado. Otra opción es conectar unos cables que proporcionen corriente a los pines Gnd y Vin de los conectores de alimentación. La propia placa es la encargada de seleccionar automáticamente la fuente de alimentación de entre estos tres.

Los límites de voltaje de entrada son 6 y 20V, aunque lo recomendable es entre 7 y 12V (con un mínimo de 7V te aseguras de que los pines de 5V suministran 5V y no menos y con un máximo de 12V evitas sobrecalentamientos en la placa).

Los pines de alimentación de la placa son los siguientes (de izquierda a derecha):

- IOREF: Pin que proporciona la referencia de voltaje con la que opera el microcontrolador (3.3V o 5V). Se utiliza para que los *shields* o extensiones para la placa se adapten al voltaje de ésta.
- 3.3V: Fuente de voltaje a 3.3 voltios y 50mA que se genera mediante un chip FTDI integrado en la placa.
- 5V: Fuente de voltaje de 5 voltios y 40mA.
- GND: Pines de toma de tierra (0V) de la placa.
- Vin: Pin a través del cual se puede proporcionar alimentación a la placa.

Entradas/Salidas

Cada uno de los pines digitales de la placa puede ser utilizado como entrada y como salida operando a 5V y proporcionando/recibiendo 20mA como corriente recomendada. El valor máximo de corriente es de 40mA, a partir del cual se provocarían daños al microcontrolador de forma permanente. Los pines también cuentan con una resistencia de *pull-up* (desconectada por defecto) de 20-50K Ohm.

Algunos pines tienen funciones especiales:

- Serie: 0 (RX0) y 1 (TX0); 19 (RX1) y 18 (TX1); 17 (RX2) y 16 (TX2); 15 (RX3) y 14 (TX3). Se utilizan para recibir (RX) y transmitir (TX) datos serie TTL. La serie 0 (pines 0 y 1) están a su vez conectado al microcontrolador ATMega16U2.
- Interrupciones externas: Las interrupciones 0, 1, 2, 3, 4 y 5 van ligadas a los pines 2, 3, 21, 20, 19 y 18, respectivamente. Estos pines se pueden configurar para lanzar una interrupción según se desee en estos casos: *LOW* (Cuando el valor es bajo), *CHANGE* (cuando cambia el valor), *RISING* (cuando el valor pasa de bajo a alto) y *FALLING* (cuando el valor pasa de alto a bajo).
- *PWM*: Pines 2-13 y 44-46. Proveen de una salida *PWM* de 8 bits (salida analógica).
- *SPI*: 50 (SS), 51 (MOSI), 52 (MISO), 53 (SCK). Pines que proporcionan la comunicación *SPI (Serial Peripheral Interface)* utilizando una librería.
- LED: El pin 13 tiene un LED integrado que se enciende cuando se le proporciona un valor alto al pin y se apaga cuando se le proporciona un valor bajo.
- *TWI*: Pines 20 (SDA) y 21 (SCL). Soportan la comunicación *TWI/I2C* utilizando una librería.

La placa también cuenta con 16 pines analógicos de entrada con una resolución de 10 bits que por defecto miden desde 0V a 5V pero se puede cambiar el valor superior mediante el pin *AREF*.

Resumen

Microcontrolador	ATMega2560
Voltaje de funcionamiento	5V
Voltaje de entrada	7-12V
Voltaje de entrada (límite)	6-20V
Pines E/S digitales	54 (15 de ellos *PWM*)
Entradas analógicas	16
Intensidad por pin de E/S	20 mA
Intensidad para el pin de 3.3V	50 mA
Memoria *Flash*	256 KB (8 KB para el *bootloader*)
SRAM	8 KB
EEPROM	4 KB
Velocidad del reloj	16 MHz

Tabla 1: Especificaciones de la placa Arduino Mega 2560

Conexión USB

La placa tiene conexión USB a través del puerto serie gracias al microcontrolador ATMega16U2, el cual además es programable, que hace de puente entre USB y el puerto serie RS-232 del microcontrolador principal ATMega2560.

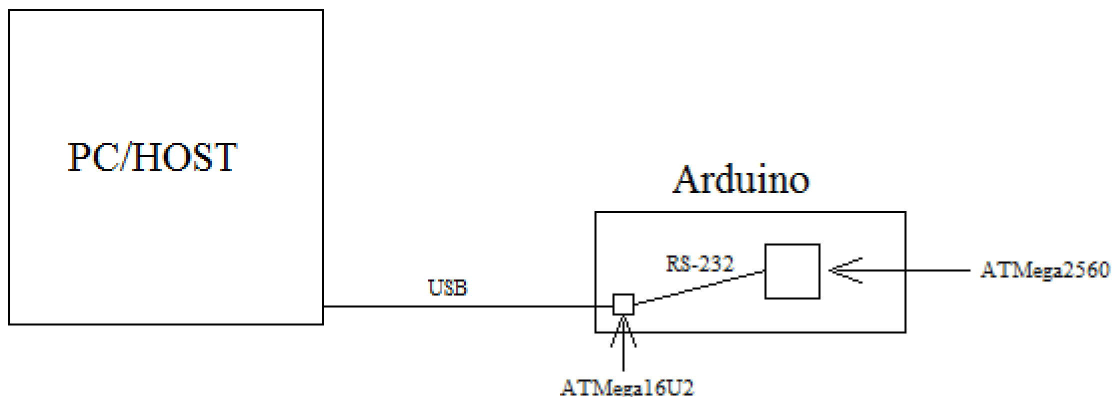

Ilustración 2: Conexión entre Arduino y un ordenador

Con la finalidad de programar el firmware del ATMega16U2 necesitaremos ponerlo en modo DFU (Device Firmware Update). Para ello se debe de hacer y luego quitar un puente entre los pines de *reset* y de *GND* de este microcontrolador, los cuales están señalados sobre la Ilustración 1.

Para salir del modo DFU basta con desconectar el cable USB y volver a conectarlo.

3.1.2 Pulsador

El pulsador, del inglés *Push button*, no es más que un interruptor que conecta dos puntos en un circuito cuando es pulsado. En este proyecto va a ser bastante utilizado y siempre va a ir ligado a una resistencia de 10K Ohm.

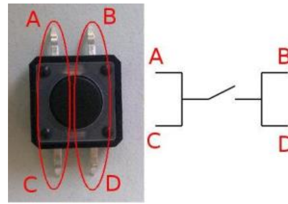

Ilustración 3: Pulsador

3.1.3 Potenciómetro

El potenciómetro es una resistencia cuyo valor es variable. En el proyecto su utilidad será la de controlar alguna variable que necesitamos que tenga más de dos posibles valores, es decir, para leer el valor analógico (hasta 1024 posibles valores) en vez del digital (2 posibles valores). Consta de 3 pines: el central conectado a una entrada analógica de Arduino u otro dispositivo y los de los extremos conectadas a Vcc y Gnd

Ilustración 4: Potenciómetro de rotación

3.1.4 *Joystick*

El *joystick* no es más que la unión de dos potenciómetros, colocados en una palanca (controlando los valores de cada uno de los dos ejes) y por un pulsador, accionado al apretar la palanca. Cuenta con cinco pines: un pin para Vcc, un pin para Gnd, un pin para el pulsador y los otros dos pines para el valor de los dos ejes.

Ilustración 5: Joystick

3.1.5 *Display* 7 segmentos

Se trata de un dispositivo capaz de representar los números del 0 al 9 controlando la iluminación de cada uno de sus 7 segmentos (además de un punto decimal). Los hay de ánodo común y de cátodo común. La principal diferencia es que los de ánodo común tienen una patilla que debe ser conectada a potencial positivo y para encender un segmento se necesita aplicar potencial negativo mientras que en los de cátodo común sucede justo al contrario, es decir, tiene una patilla que debe conectarse a potencial negativo y los segmentos que se quieran encender debe de aplicarse sobre ellos un potencial positivo. En nuestro caso utilizaremos el de cátodo común.

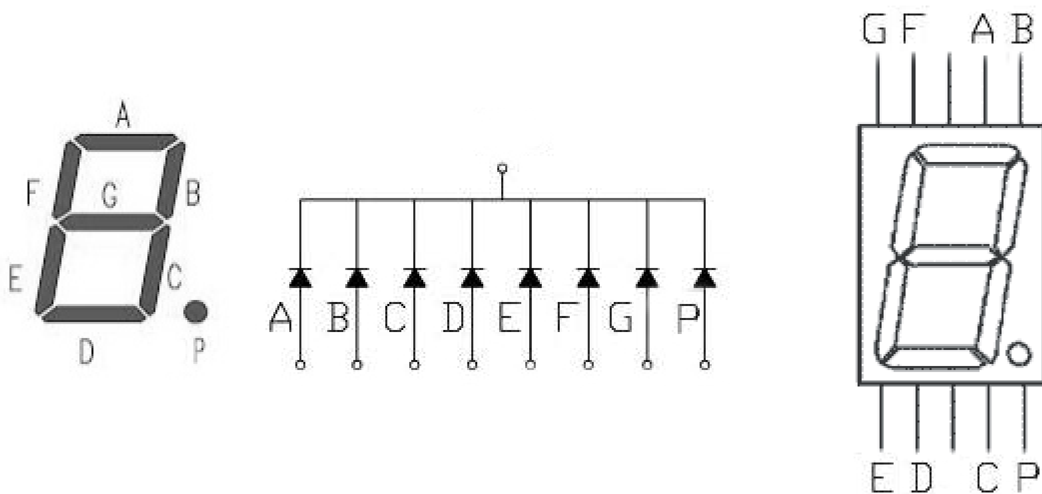

Ilustración 6: Display de 7 segmentos de cátodo común

3.1.6 Sensor de ultrasonidos HC-SR04

Ilustración 7: Sensor de ultrasonidos HC-SR04

El sensor de ultrasonidos es un detector de proximidad cuyo funcionamiento consiste en enviar una señal y medir el tiempo que tardar ésta en volver. Si el objeto está cerca la señal tardará menos en rebotar y volver que si está lejos.

El modelo HC-SR04 nos permite medir unas distancias de entre 2cm y 400m con un margen de error de 3mm, midiendo distancias con un ángulo máximo de 15 grados. El funcionamiento básico de este sensor en concreto es el siguiente:

1. Aplicar un potencial alto al disparador (*Trigger pin*) durante al menos 10us
2. El módulo automáticamente envía ocho ondas a 40kHz y detecta si hay un pulso de vuelta.
3. Si se ha detectado señal de vuelta, el tiempo en el que permanece en alto potencial la salida (*Echo pin*) es el tiempo que ha tardado el ultrasonido en ir y volver.

La fórmula para obtener la distancia quedaría así:

(tiempo en alto potencial del pin *Echo* x velocidad del sonido)/2.

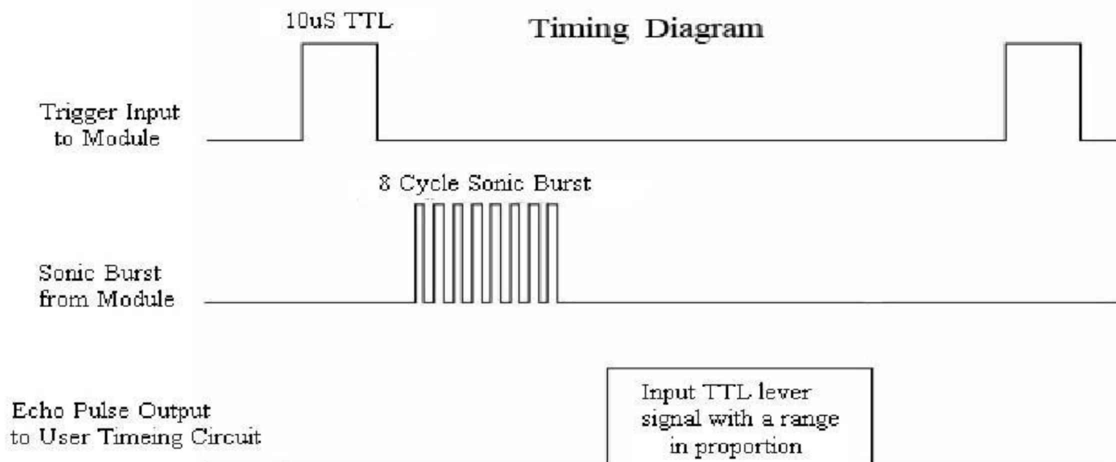

Ilustración 8: Diagrama de tiempo del sensor de ultrasonidos HC-SR04

3.1.7 *Display* LCD 16x2

El *display* LCD es una pantalla de 2 líneas con 16 caracteres por línea, siendo cada carácter representado en una matriz de 5x8 píxeles. Está compuesto por dos registros: el registro de comandos (guarda los comandos enviados al LCD, como pueden ser limpiar la pantalla o mover el cursor) y el registro de datos (guarda los datos que tienen que ser mostrados en el *display*). A continuación se muestra una imagen con los pines y una tabla con sus funciones:

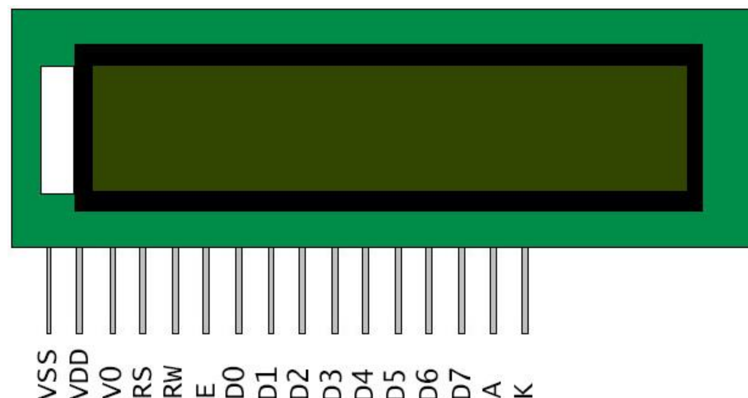

Ilustración 9: Display LCD 16x2

Nº Pin	Nombre	Función
1	V_{SS}	Toma de tierra
2	V_{DD}	Voltaje de funcionamiento (5V)
3	V_0	Ajuste de contraste (a través de una resistencia variable)
4	RS	Señal de selección de registro (Alto: comandos; Bajo: datos)
5	RW	Señal de lectura/escritura (Alto: lectura; Bajo: escritura)
6	E	Señal de habilitación (Se lee/escribe en los flancos de bajada)
7	D0	4 bits de orden bajo del bus de datos. Estos no se utilizan en las operaciones de 4 bits
8	D1	
9	D2	
10	D3	
11	D4	4 bits de orden alto del bus de datos. Se utilizan tanto en las operaciones de 4 bits como en las de 8 bits
12	D5	
13	D6	
14	D7	
15	A	V_{CC} de la iluminación de fondo
16	K	V_{SS} de la iluminación de fondo

Tabla 2: Descripción de los pines del display LCD

3.1.8 Teclado 4x4

Ilustración 10: Teclado 4x4

El teclado 4x4 es un dispositivo que, como su propio nombre indica, cuenta con una disposición de teclas de una matriz de cuatro columnas y cuatro filas, haciendo un total de dieciséis teclas formadas por pulsadores. En total cuenta con 9 pines: uno para cada columna, uno para cada fila, y uno para VCC. Cada fila se encuentra conectada a una resistencia de *pull up*.

El funcionamiento para averiguar qué tecla está siendo pulsada es el siguiente:

```
(Por defecto los pines de fila deben tener potencial alto)
Para cada fila:
    Aplicar potencial bajo en la fila.
    Para cada columna:
        Si la columna tiene potencial bajo:
            La tecla pulsada corresponde a la columna y fila de esta iteración.
    Aplicar potencial bajo en la fila.
```

Ilustración 11: Pseudocódigo para obtener la tecla que está siendo pulsada

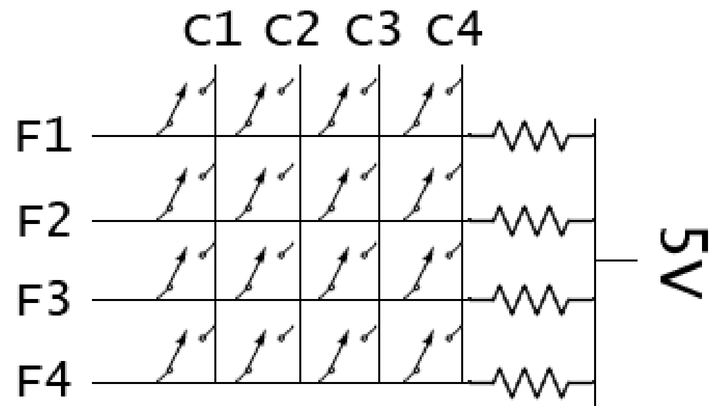

Ilustración 12: Esquema de un teclado 4x4

3.2 *Software*

3.2.1 Arduino IDE

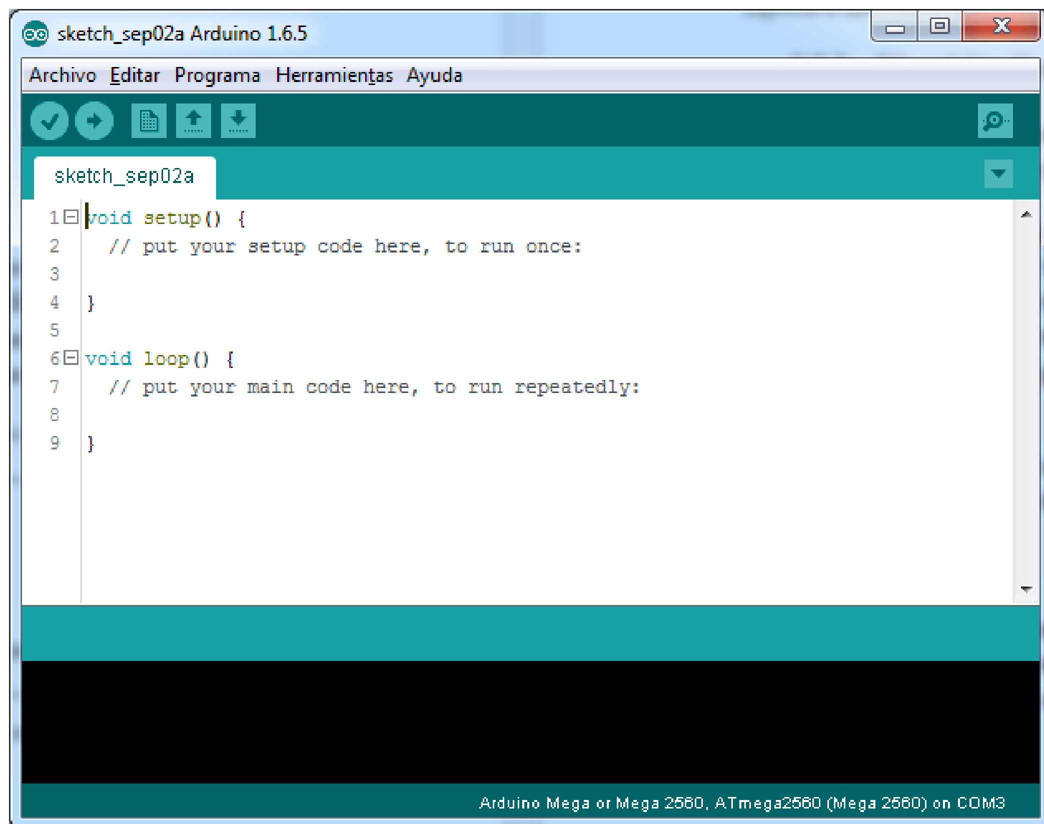

Ilustración 13: Interfaz gráfica de Arduino IDE

Arduino IDE es el entorno de desarrollo oficial para Arduino. Se trata de *software* de código libre que está disponible tanto para Windows, para Linux y para Mac OS X. La herramienta es muy básica; está compuesta por un editor de texto bastante simple y cuenta con opciones de compilación y subida del *sketch* (programa) a la placa, siempre y cuando ésta se encuentre conectada al ordenador.

No obstante, con la instalación de la herramienta se instalan los controladores para que nuestro ordenador reconozca la placa, las librerías de Arduino, el set de herramientas de gcc y el *software* necesario para subir el código compilado a la placa.

3.2.2 Visual Studio 2013 (CE) + Visual Micro

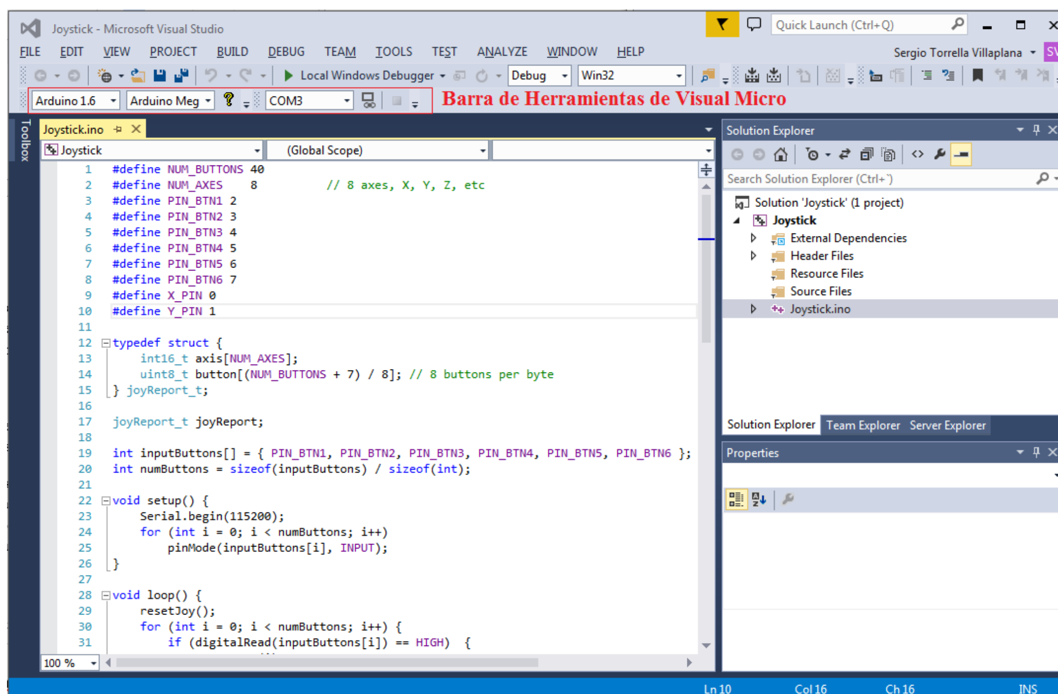

Ilustración 14: Visual Studio 2013 con Visual Micro, un plugin para Arduino

Visual Studio es un entorno propietario de Microsoft únicamente disponible en sistemas operativos Windows que soporta múltiples lenguajes de programación.

Visual Micro es un *plugin* para Visual Studio que añade la funcionalidad de poder programar para Arduino. Se ha optado por este IDE dado que es un editor mucho más avanzado y con más funcionalidades (tales como el autocompletado o la detección de sintaxis errónea, así como sus atajos de teclado) que sirven de ayuda a una programación más rápida y eficiente. Visual Micro también cuenta con un *Debugger* (de pago) que sirve de ayuda a cazar errores en el programa en tiempo real.

Necesita tener instalado Arduino IDE para poder utilizar sus librerías, su set de herramientas de gcc y el *software* para subir el código compilado a la placa.

3.2.3 Fritzing

Fritzing es un programa de código libre que está basado en los mismos principios de Processing y Arduino. Permite a los usuarios documentar sus prototipos, compartirlos con los demás, enseñar electrónica en clase y crear pcbs profesionales.

De todas las funcionalidades de este programa, únicamente se ha explotado la parte de documentar los diseños, creando el esquema de conexiones entre los distintos componentes/dispositivos y la placa de Arduino.

3.2.4 Flip

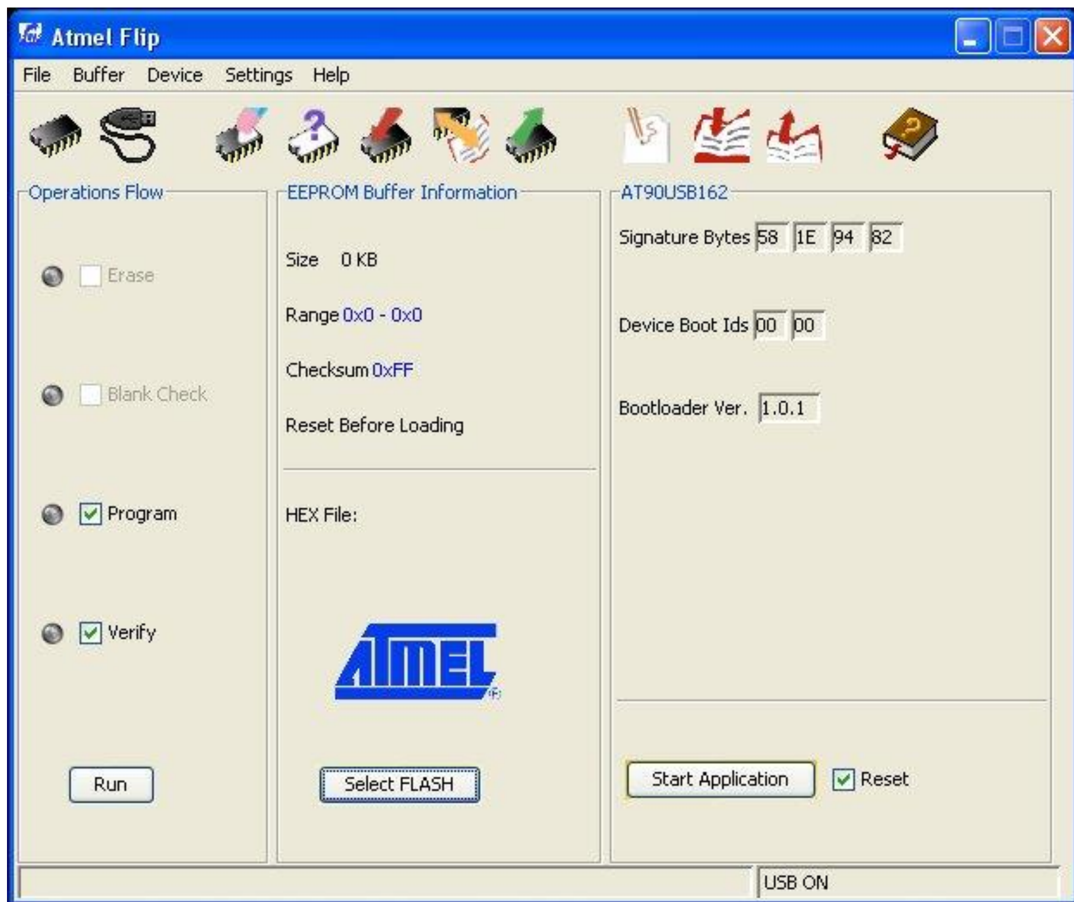

Ilustración 15: Interfaz gráfica de Flip

Flip es un *software* creado por Atmel que nos permite programar el *firmware* del microcontrolador que hace de puente entre el microcontrolador de Arduino y el ordenador; en nuestro caso el ATMega16U2.

Para realizar este procedimiento debemos poner el microcontrolador en modo DFU para que Flip lo reconozca y acto seguido cargar el *firmware* (archivo .hex) y programar el ATMega16U2.

4. Dispositivos desarrollados

En esta sección se desarrolla una serie de dispositivos HID construyendo el *hardware* y el *software* que lo controla en la plataforma Arduino. Los drivers HID para poder utilizarlos como periféricos en un *host* los proporciona el *firmware* que programemos en el microcontrolador ATMega16U2, utilizando en esta sección unos *firmwares* desarrollados por terceros con todo lo que ello implica, es decir, hay que adaptarse al protocolo de mensajes que el desarrollador diseñó para la comunicación entre los microcontroladores ATMega2560 y ATMega16U2.

4.1 Teclado con teclas personalizables

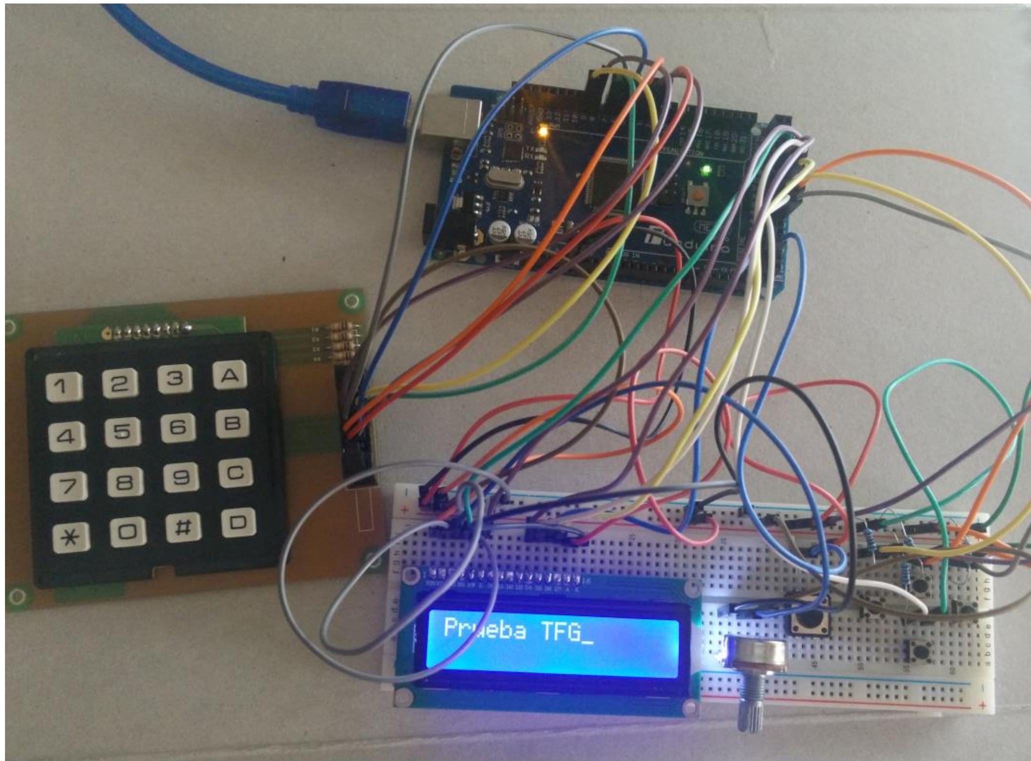

Ilustración 16: Teclado HID

Este dispositivo HID consta de un teclado de 4x4 teclas cuyo valor por defecto es el que tienen impreso. Cuando se pulsa una tecla, además de enviar el código específico al *host*, se muestra el valor de esta tecla en la pantalla LCD (Exceptuando las teclas especiales). Mediante cuatro botones y con la ayuda de la pantalla LCD se permite que el usuario cambie el valor de cualquiera de las dieciséis teclas, guardándose en la EEPROM de manera que la configuración permanece a pesar de que se reinicie el dispositivo. También cuenta con un botón de *reset* para devolver el valor por defecto a las teclas. El contraste de la pantalla LCD se regula mediante un potenciómetro.

Los valores que se pueden asignar a las teclas son las siguientes: los caracteres alfanuméricos (a-z, A-Z, 0-9), la coma (,), el punto (.), el cierre de exclamación (!), el cierre de interrogación (?), el asterisco (*), la almohadilla (#), el espacio ( ), el carácter especial de borrado y las funciones de cortar (Ctrl+x), copiar (Ctrl+c) y pegar (Ctrl+v).

Material necesario

- 1 x Arduino Mega 2560
- 1 x Matriz de 4 x 4 teclas con resistencias de *pull up*
- 1 x *Display* LCD de 2 filas y 16 columnas.
- 1 x Potenciómetro
- 5 x Pulsadores/Botones
- 5 x Resistencias de 10 kOhms.
- Cables para realizar las conexiones

Esquema de montaje

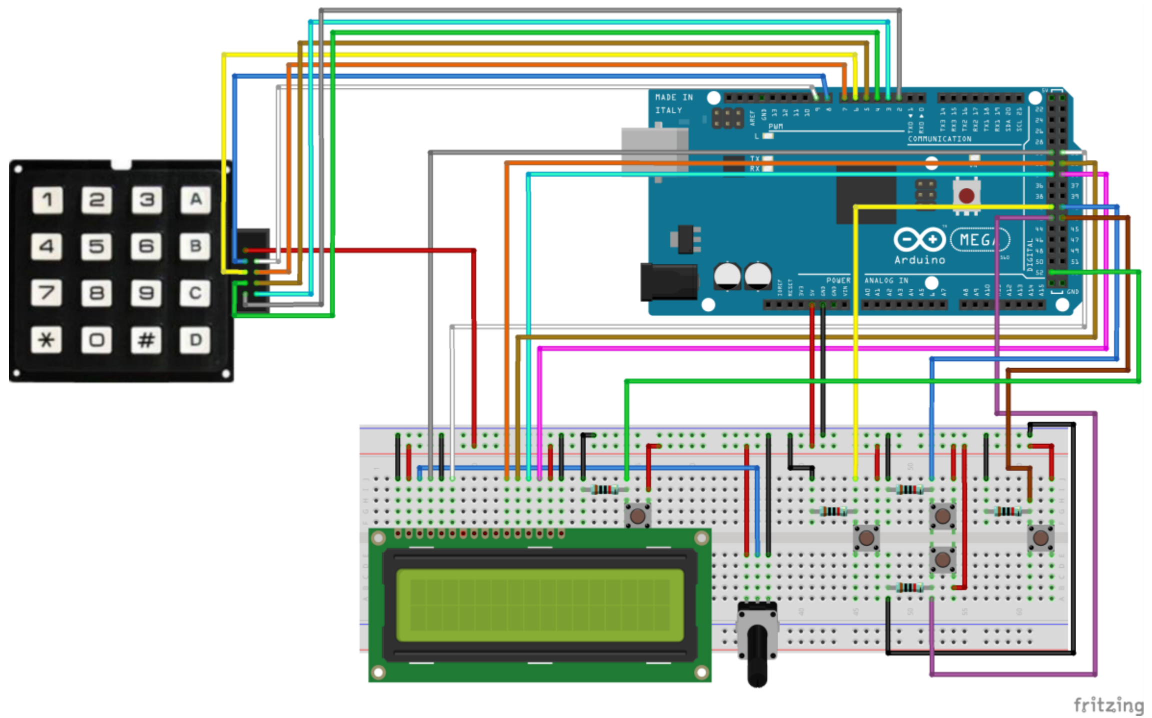

Ilustración 17: Esquema de montaje del teclado HID

Funcionalidad de los componentes

Componente	Función
Teclas del teclado 4x4	-En modo normal envía el valor de la tecla al *host*. -En modo configuración selecciona la tecla cuyo valor se va a cambiar.
Potenciómetro	Regula el contraste de la pantalla LCD.
Botón de *reset*	Si se pulsa durante 3 o más segundos reinicia la configuración por defecto de las teclas (el valor que éstas tienen impreso). Sólo funciona en modo normal.
Botón izquierdo	Alterna el modo: -En modo normal pasa a modo configuración. -En modo configuración pasa a modo normal.
Botón superior	Botón ascendente de navegación para seleccionar el valor de una tecla. Sólo útil en modo configuración.
Botón inferior	Botón descendente de navegación para seleccionar el valor de una tecla. Sólo útil en modo configuración.
Botón derecho	Confirma el nuevo valor de la tecla seleccionada. Sólo útil en modo configuración.
Pantalla LCD	-En modo normal muestra el valor de las teclas que se van pulsando. -En modo configuración sirve de ayuda para cambiar el valor de las teclas, ofreciendo una interfaz al usuario.

Tabla 3: Función de cada uno de los componentes del teclado HID

El programa cuenta con dos modos de funcionamiento. Uno de ellos es el propio modo de teclado, donde la tecla que se pulsa es enviada al *host* y, a su vez, mostrada en el *display*. El otro modo es el de configuración (accesible al pulsar el botón izquierdo) donde se nos solicita pulsar la tecla cuyo valor deseamos cambiar. Podemos navegar por los distintos valores con los botones de arriba y abajo y, cuando tengamos el nuevo valor deseado en pantalla, pulsamos el botón derecho para guardar los cambios y salir del modo de configuración. Si en vez de pulsar el botón derecho pulsamos el izquierdo, sale del modo de configuración directamente sin guardar los cambios.

El botón de *reset* tiene la función de restaurar el valor por defecto de los botones (1 2 3 A; 4 5 6 B; 7 8 9 C; * 0 # D) si lo mantenemos pulsado durante tres segundos en el modo de funcionamiento normal.

El funcionamiento del programa queda resumido en el diagrama de flujo del teclado HID (Ilustración 18).

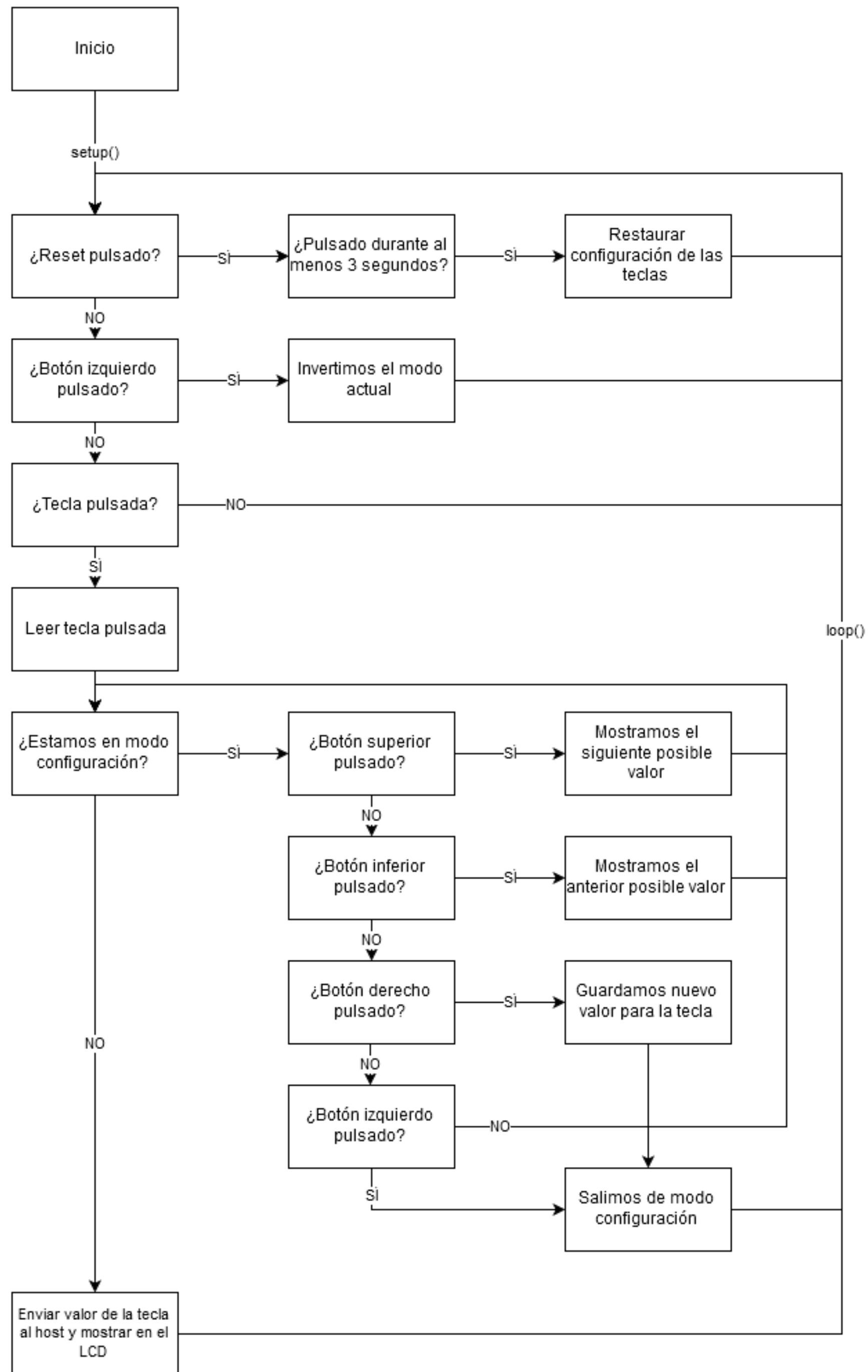

Ilustración 18: Diagrama de flujo del teclado HID

A pesar de haber librerías en Arduino para controlar las matrices de teclas, se ha optado por implementar una función para obtener la tecla que está siendo pulsada:

```
int getKey()
{
   long startTime = millis();
   //Evitamos falsos positivos leyendo únicamente cuando ha pasado el tiempo de "rebote" 
desde la última lectura
   if (startTime - lastTimeKeyPressed > KEY_DEBOUNCE_TIME)
   {
      //Buscamos teclas pulsadas durante KEY_TIMEOUT ms, si no devolvemos -1
      while (millis() <= startTime + KEY_TIMEOUT)
      {
         for (int i = 0; i < 4; i++)
         {
            //Aplicamos potencial bajo a la fila i
            digitalWrite(rows[i], LOW);
            for (int j = 0; j < 4; j++)
            {
               //Si leemos potencial bajo en la columna j devolvemos el indice de la tecla 
[i,j] (De 0 a 15)
               if (digitalRead(cols[j]) == LOW)
               {
                  lastTimeKeyPressed = millis();
                  if (!isKeypadPressed[i][j])
                  {
                     isKeypadPressed[i][j] = true;
                     digitalWrite(rows[i], HIGH);
                     return i * 4 + j;
                  }
               }
               else
               {
                  isKeypadPressed[i][j] = false;
               }
            }
            //Restauramos potencial alto a la fila i
            digitalWrite(rows[i], HIGH);
         }
      }
   }
   return -1;
}
```

Código 1: Función para obtener la tecla que está siendo pulsada

Para el control de la pantalla LCD sí que se ha decidido utilizar la librería *LiquidCrystal*:

```
#include <LiquidCrystal.h>
…
LiquidCrystal lcd(PIN_RS, PIN_E, PIN_D4, PIN_D5, PIN_D6, PIN_D7);
…
lcd.begin(16, 2);   //16 columnas x 2 filas
lcd.cursor();
…
lcd.noCursor();
lcd.setCursor(0, 0);
lcd.print("Pulsa una tecla");
lcd.setCursor(0, 1);
lcd.print("para configurar");
…
lcd.clear();
…
```

Código 2: Ejemplos de uso de la librería LiquiCrystal dentro del código del teclado HID

4.2 Teclado musical MIDI

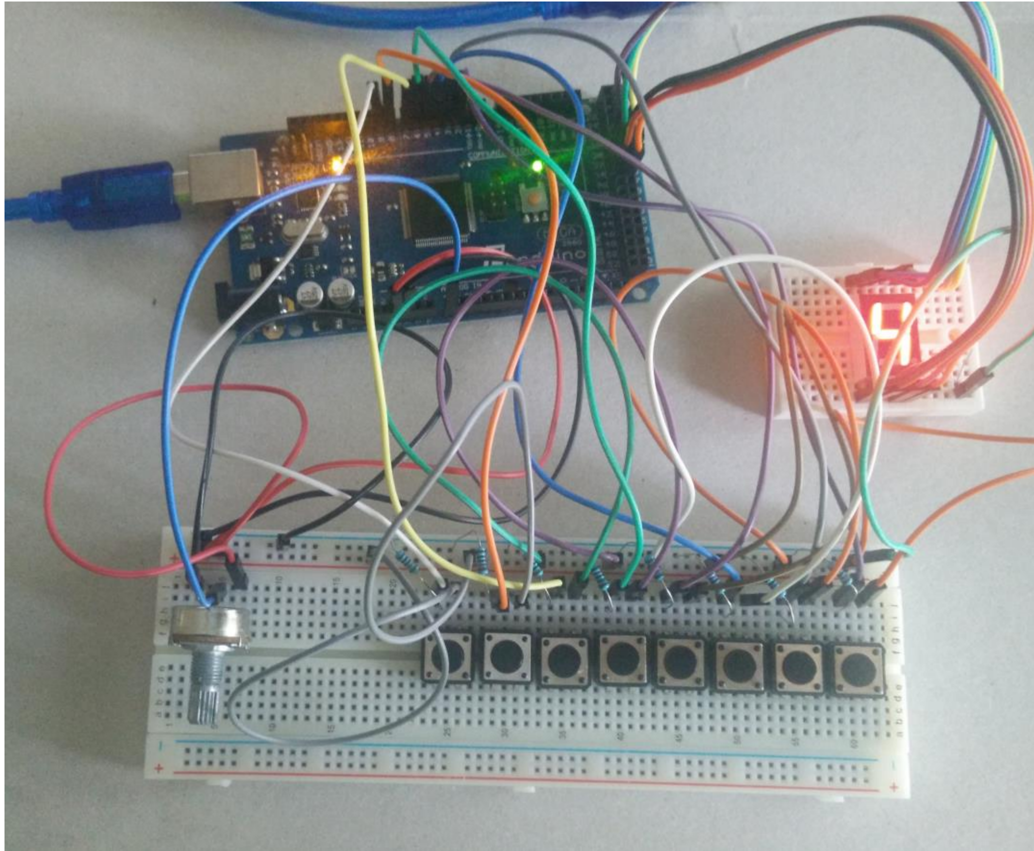

Ilustración 19: Teclado musical MIDI

El teclado musical MIDI es un dispositivo HID de entrada en el que nos encontramos una serie de botones o teclas que representan la escala de diatónica notas (do-re-mi-fa-sol-la-si-do) dentro de una octava, pudiendo variar entre 7 diferentes octavas con la ayuda de un potenciómetro. Podemos ver el número de octava en el que estamos a través de un *display* de 7 segmentos.

Material necesario

- 1 x Arduino Mega 2560
- 1 x *Display* de 7 segmentos.
- 1 x Potenciómetro
- 8 x Pulsadores/Botones
- 8 x Resistencias de 10 kOhms.
- 1 x Resistencia de 220 Ohms.
- Cables para realizar las conexiones

Esquema de montaje

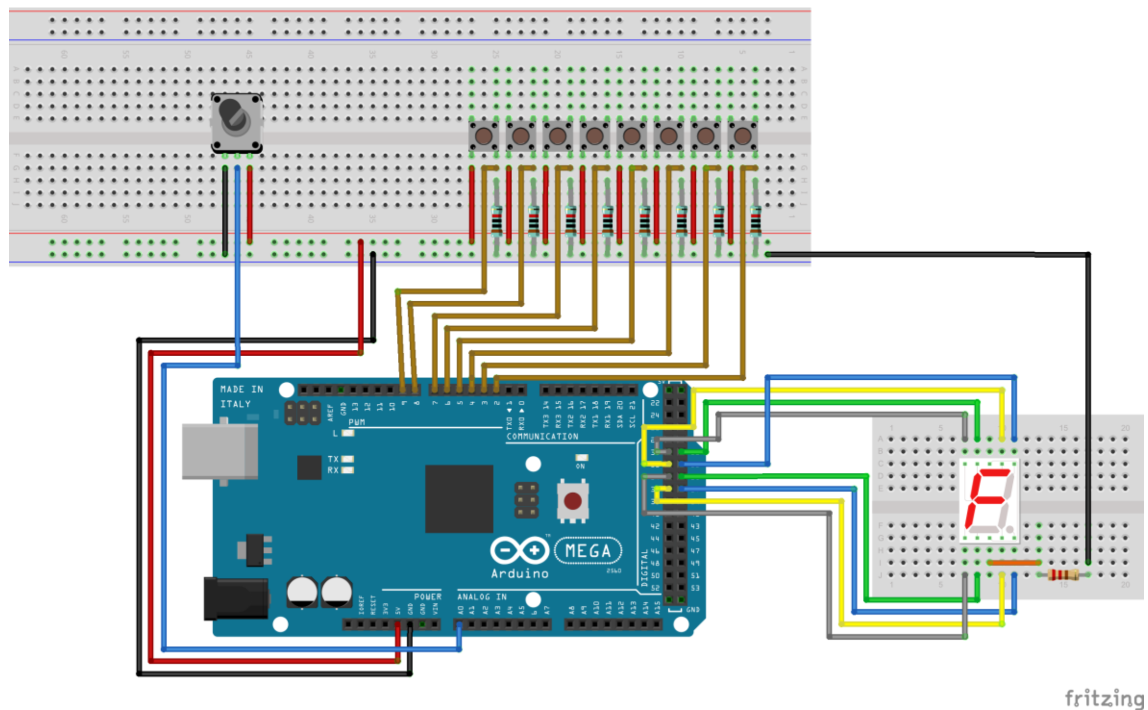

Ilustración 20: Esquema de montaje del teclado MIDI

Funcionalidad de los componentes

Componente	**Función**
Botones/teclas	-Representan, de izquierda a derecha, las notas do-re-mi-fa-sol-la-si-do. -Al pulsarlo (se pueden pulsar varios a la vez) se envía la pulsación de la nota correspondiente al *host*. -Al soltarlo se envía al *host* que la nota correspondiente se ha soltado.
Potenciómetro	Regula el número de octava en el que suenan las notas. Los números de octava van del 1 al 7, siendo la más grave la 1 y la más aguda la 7
Display 7 segmentos	Se encarga de reflejar el número de octava en el que se está

Tabla 4: Función de cada uno de los componentes del teclado MIDI

El programa que controla el teclado MIDI es bastante sencillo dado que únicamente se encarga de calcular la octava actual mediante el potenciómetro, actualizar el *display* si la octava ha cambiado y enviar al *host* los cambios producidos, es decir, si se ha pulsado o soltado alguna tecla respecto a la iteración anterior. Para el envío de estos cambios se ha utilizado la librería MIDI.

La parte más complicada ha sido controlar el rebote de las teclas. El rebote de un pulsador es un efecto mecánico mediante el cual, al soltar el pulsador, se produce un instante de tiempo en el que se producen pulsaciones fantasma, es decir, se perciben cambios de potencial alto a bajo y viceversa en la lectura. La forma de evitar este efecto es definir un tiempo tras la pulsación o el soltado de una tecla en el que ignoramos la entrada.

```
    //Para cada tecla
    for (int i = 0; i < 8; i++)
    {
       //Si han pasado los milisegundos de rebote desde el último cambio de la tecla
       if (millis() - lastToggledMillis[i] > DEBOUNCE_TIME)
       {
          //Si la tecla está pulsada y en la iteración anterior no lo estaba
          if (digitalRead(inputButtons[i]) == HIGH && !keyPressed[i])
          {
             //Actualizamos variables y enviamos el código correspondiente a la nota que se ha
pulsado (según tecla y octava)
             lastToggledMillis[i] = millis();
             MIDI.sendNoteOn(24 + 12 * (level - 1) + notes[i], 120, 1);
             keyPressed[i] = true;
          }
          //Si la tecla no está pulsada y en la iteración anterior si lo estaba
          else if (digitalRead(inputButtons[i]) == LOW && keyPressed[i])
          {
             //Actualizamos variables y enviamos el código correspondiente a la nota que se ha
soltado (según tecla y octava)
             lastToggledMillis[i] = millis();
             MIDI.sendNoteOff(24 + 12 * (level - 1) + notes[i], 120, 1);
             keyPressed[i] = false;
          }
       }
    }
```

Código 3: Código de lectura de cambios en las teclas, teniendo en cuenta el tiempo de rebote de éstas

Otro caso especial en el funcionamiento del dispositivo se produce cuando se cambia de octava mientras se pulsa una tecla. Sin tratar esto, obtendríamos como resultado que la nota que hubiera pulsada en ese momento permanecería pulsada hasta que se volviera a esa octava y se pulsara y soltara otra vez la tecla correspondiente. Para evitar este comportamiento anómalo se ha realizado un tratamiento de la situación que consiste en que, cuando se cambie de octava, se comprueba si había alguna tecla pulsada y en caso afirmativo se envía el mismo comando que si se soltara la nota correspondiente a esa tecla y la octava anterior.

4.3 Ratón de ultrasonidos

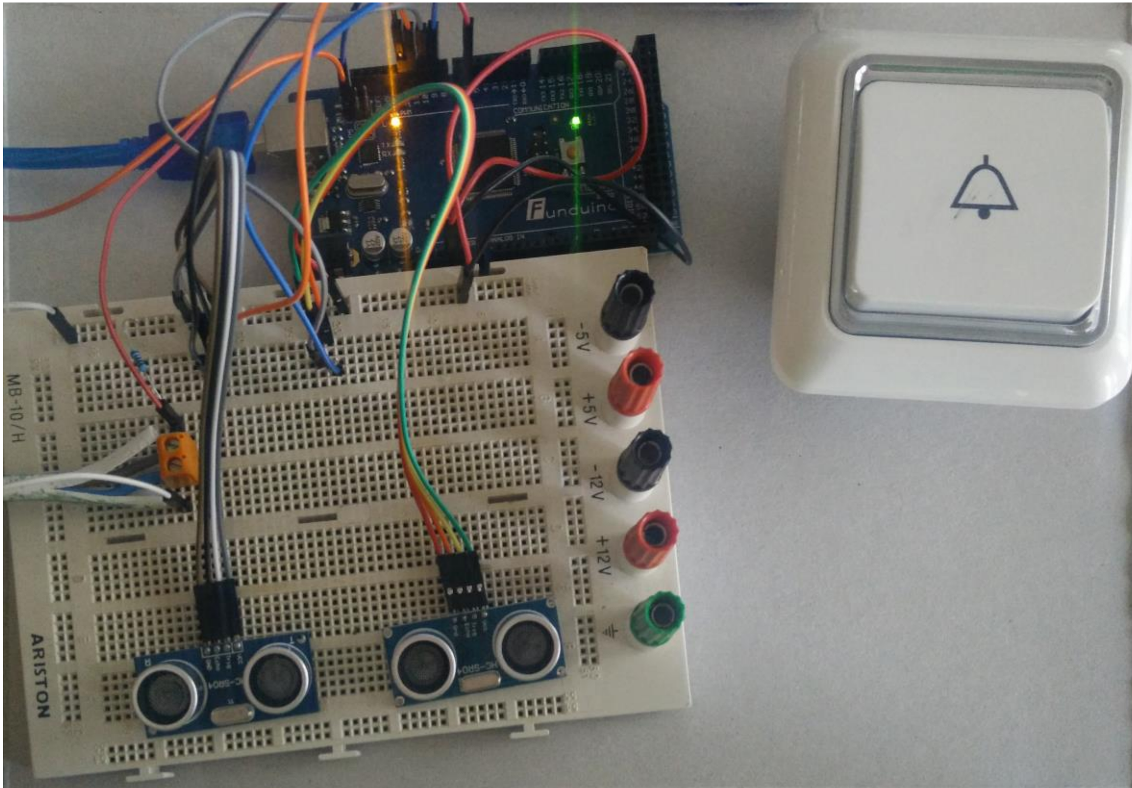

Ilustración 21: Ratón HID de ultrasonidos

El ratón HID de ultrasonidos está formado con dos dispositivos de ultrasonidos que mide la distancia que hay hacia un objeto para así calcular el movimiento del objeto y, en consecuencia, mover cada uno de los ejes del puntero del ratón sobre la pantalla. Cuenta con un solo botón para hacer *click* (en mi caso se ha utilizado el pulsador de un timbre con la finalidad de accionarlo con el pie), que correspondería al botón izquierdo de los ratones convencionales.

Material necesario

- 1 x Arduino Mega 2560
- 2 x Sensores ultrasónicos HC-SR04
- 1 x Pulsador/Botón
- 1 x Resistencia de 10 kOhms.
- Cables para realizar las conexiones

Esquema de montaje

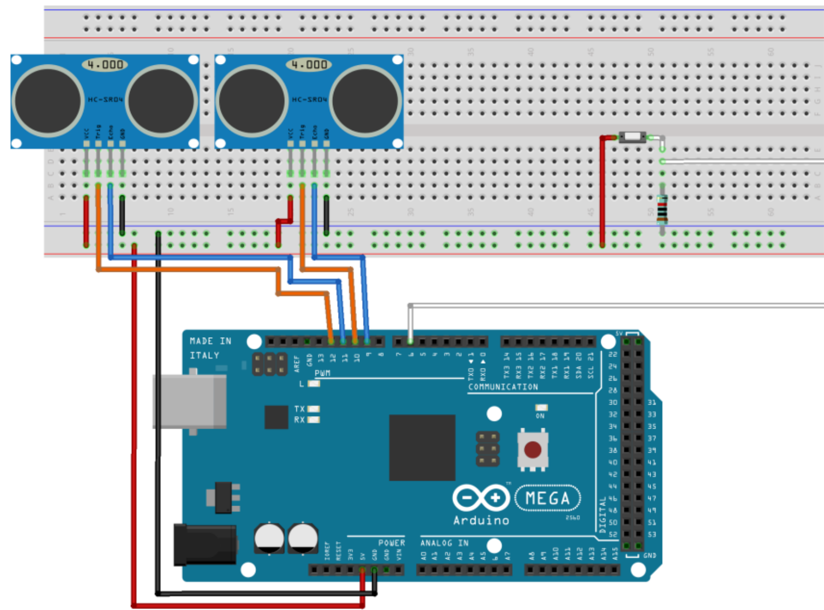

Ilustración 22: Esquema de montaje del ratón de ultrasonidos

Funcionalidad de los componentes

Componente	Función
Botón	Representa el botón izquierdo de un ratón convencional
Sensores de ultrasonido HC-SR04	Miden la distancia hacia un objeto (normalmente la mano) en cada instante para calcular la diferencia y mover el eje correspondiente

Tabla 5: Función de cada uno de los componentes del ratón de ultrasonidos

El programa que controla el ratón consiste en, únicamente, leer el estado de los botones y los sensores de ultrasonido y enviarlos al *host*. Lo más complicado ha sido traducir la lectura de los sensores a valores que hagan que el ratón se mueva como nosotros queramos. Para controlar los HC-SR04 se ha utilizado la librería Ultrasonic, con la ayuda de una clase "Axis" que se ha desarrollado para evitar la repetición innecesaria de código. A continuación se incluye el cógido de esta clase con comentarios para entenderlo.

```
#include <Arduino.h>
#include <Ultrasonic.h>

class Axis : public Ultrasonic {
public:
    Axis(int triggerPin, int echoPin);
    int calculateMovement();

private:
    int m_lastTime;     //Variable para guardar el último tiempo leído
};
```

```
#include "Axis.h"

// Constructor con el pin de trigger y de echo para llamar al constructor de Ultrasonic

Axis::Axis(int triggerPin, int echoPin) : Ultrasonic(triggerPin, echoPin)
{
    m_lastTime = 0;
}

// Función para obtener el movimiento que ha habido desde la última vez que se llamó a la
función. En caso de no detectar movimiento o exceder unos límites se devuelve 0

int Axis::calculateMovement()
{
    // Se lee el tiempo que tarda la onda en llegar al objeto y volver a al pin de echo
    int timing = this->Timing();
    int timeDiff = 0;

    // Comprobación de ciertos límites en la lectura
    // Los límites han sido obtenidos de forma empírica. A partir de estos los valores no son
del todo fiables
    if (timing < 2700 && timing > 300 && m_lastTime < 2700 && m_lastTime > 300)
    {
        //Calculamos la diferencia de tiempo respecto a la vez anterior
        timeDiff = timing - m_lastTime;
        //No permitimos una diferencia mayor a 1500
        timeDiff = constrain(timeDiff, -1500, 1500);
        //El valor devuelto a devolver debe ser de un byte, por lo que hacemos el map para
obtener un máximo de 256 posibles valores
        timeDiff = map(timeDiff, -1500, 1500, 128, -127);
    }
    m_lastTime = timing;

    return timeDiff;
}
```

Código 4: Código de la clase "Axis", utilizada para controlar un eje a través de un sensor de ultrasonidos

El creador del *firmware* que da la funcionalidad HID ha implementado un protocolo consistente en enviar 4 bytes con el siguiente contenido:

Byte	Descripción
0	Estado de los botones: 0 no pulsado, 1 pulsado Bit 0 – Botón 1, Bit 1 – Botón 2, Bit 2 – Botón 3
1	Movimiento del eje X: int8_t tango de -127 a +128
2	Movimiento del eje Y: int8_t tango de -127 a +128
3	Movimiento de la rueda. Todavía no implementado

Tabla 6: Protocolo de comunicación para el ratón de ultrasonidos

A su vez, tras indicar el movimiento de los ejes al *host*, se debe enviar otro paquete con los bytes 1 y 2 con valor 0 para resetear los ejes y que en el próximo mensaje el *host* tome como referencia el reposo, y no el movimiento que ya llevaban los ejes.

El siguiente extracto de fragmentos de código refleja el paso de mensajes siguiendo el protocolo del *firmware*:

```
typedef struct{
   uint8_t buttons;
   int8_t x;
   int8_t y;
   int8_t wheel;
} Report;

Report mouseReport, nullReport;
…
void loop()
{
   //Rellenamos mouseReport
   if (digitalRead(LEFT_BUTTON_PIN) == CLICKED)
      mouseReport.buttons = 1;
   else
      mouseReport.buttons = 0;

   mouseReport.x = xAxis.calculateMovement();
   mouseReport.y = yAxis.calculateMovement();
   mouseReport.wheel = 0;
   //Enviamos el estado del botón y los ejes
   Serial.write((uint8_t *)&mouseReport, 4);
   nullReport.buttons = mouseReport.buttons;
   //Enviamos un valor 0 para los ejes manteniendo el estado del botón
   Serial.write((uint8_t *)&nullReport, 4);

   delay(50);
}
```

Código 5: Protocolo de mensajes del ratón HID de ultrasonidos

4.4 Mando para juegos

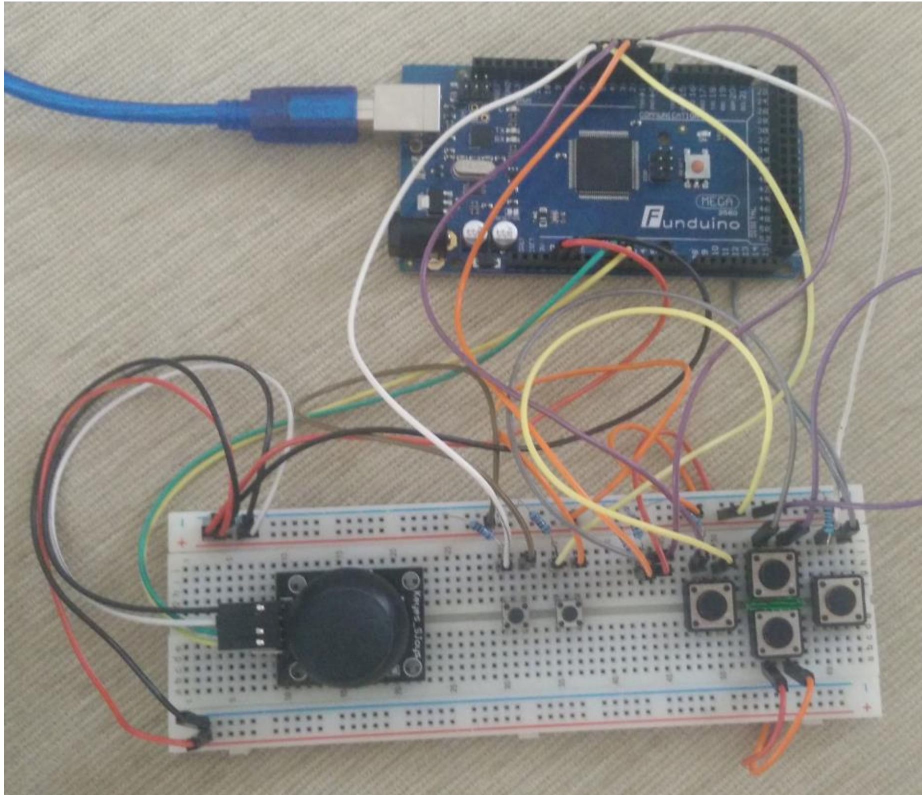

Ilustración 23: Mando para juegos HID

El mando para juegos está formado por seis botones (dos de ellos centrales y los otros cuatro a la derecha en forma de cruz) y un *joystick* analógico que controla la dirección en dos ejes.

Material necesario

- 1 x Arduino Mega 2560
- 1 x *Joystick* analógico
- 6 x Pulsadores/Botones
- 6 x Resistencia de 10 kOhms.
- Cables para realizar las conexiones

Esquema de montaje

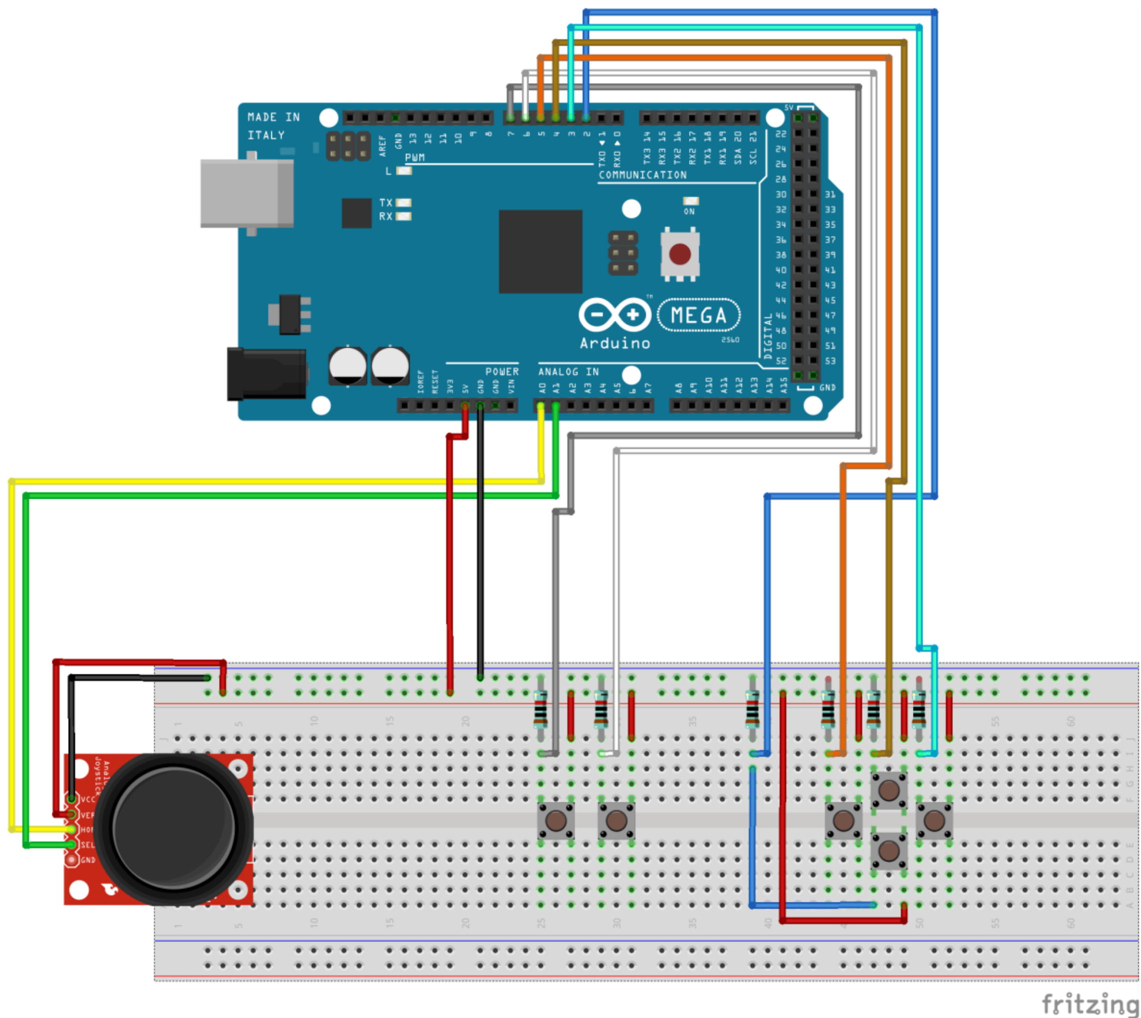

Ilustración 24: Esquema de montaje del mando para juegos

Funcionalidad de los componentes

Componente	**Función**
Botones	Representan cada uno de los seis botones de los que dispone el mando. Su funcionalidad vendrá determinada por la configuración que nosotros le queramos dar en las opciones de cada juego en particular.
Joystick analógico	Controla dos ejes de forma analógica (hasta 1024 diferentes valores, aunque el *firmware* soporta 65536). Su funcionalidad también viene determinada por la configuración de cada juego pero usualmente se utiliza para controlar el movimiento de algo (un personaje, un objeto, etc.)

Tabla 7: Función de cada uno de los componentes del mando para juegos

En este programa no ha sido necesario el uso de ninguna librería externa. Al utilizar un *firmware* ya diseñado por otro programador, hay que adaptarse a sus especificaciones, siendo necesario enviar el estado de 40 botones y 8 ejes a pesar de utilizar únicamente 6 botones y 1 eje. Los valores a enviar deben ser un 1 si el botón está pulsado y un 0 si no lo está, agrupándolos en un byte por cada 8 botones. En el caso de los ejes, se debe enviar un valor comprendido entre -32768 y 32767. Dado el valor que se lee de cada eje está comprendido entre 0 y 1023, hay que utilizar la función map() para ajustarlo a uno que cumpla la especificación. Con los valores ya adaptados a la especificación se procede al envío de los mismos por el puerto serie, teniendo que enviar un total de 20 bytes, de los cuáles se rellenan únicamente los bytes 0-3 (0 y 1 para el eje X y 1 y 2 para el eje Y) y el byte 16 (correspondiendo el bit 0 con el botón 1 y el bit 5 con el botón 6).

5. Creación de un *firmware*

Hasta ahora, se ha desarrollado varios tipos de dispositivos HID utilizando un *firmware* ya diseñado una tercera persona y ha habido que adaptar el *software* al protocolo que éste decidió emplear para la comunicación entre el microcontrolador de Arduino y el microcontrolador ATMega16u2 que hace de puente entre puerto serie y USB. Esta necesidad de adaptación al protocolo puede limitar la funcionalidad deseada o, como en el caso del mando para juegos, aportar funcionalidades que no se van a utilizar (40 botones y 8 ejes, de los que solamente se utilizan 6 y 2, respectivamente). Por tanto, se va a diseñar un *firmware* que se pueda adaptar a las necesidades específicas de este dispositivo en concreto.

5.1 Protocolo de mensajes

Se ha decidido utilizar un protocolo de mensajes donde únicamente se envían los cambios que sucedan en los botones y en los ejes. De esta forma, además de evitar saturar el puerto serie con mensajes redundantes, se demuestra que se tiene cierta flexibilidad en el diseño. El protocolo quedaría de la siguiente forma:

- **Mensaje para indicar cambios en un botón:** Se compone por dos bytes. El primer byte corresponde con el carácter ‘B’. En el segundo byte se indica tanto el número de botón que cambia (del 1 al 6) en los 4 bits de mayor peso como el valor del botón (1 = ON; 0 = OFF) en los 4 bits de menor peso. Por ejemplo, para indicar que el botón 2 está pulsado se enviaría el siguiente mensaje: Byte 1 = ‘B’; Byte 2 = 0x21 (0010 0001 en binario)
- **Mensaje para indicar cambios en un eje:** Se compone por tres bytes. El primer byte corresponde con el carácter ‘X’ si el eje que ha cambiado es el eje X y con el carácter ‘Y’ en caso de cambio del eje Y. Como los valores de estos ejes están comprendidos entre 0 y 1023 se necesita al menos 10 bits para representar todos los posibles valores, por lo que se deben utilizar dos bytes. De este modo, en el segundo byte se envían los 2 bits de mayor peso y en el tercer byte se envían los 8 bits de menor peso. Por ejemplo, para un valor del eje X = 1020 (11 1111 1100 en binario) el mensaje sería el siguiente: Byte 1 = ‘X’; Byte 2 = 0x3 (0011 en binario); Byte 3 = 0xFC (1111 1100 en binario).

5.2 Entorno de desarrollo

Para poder llevar a cabo el desarrollo de este *firmware* han sido necesarias las siguientes herramientas:

- Entorno Linux (Bien a través de SO Linux o bien en Windows con herramientas como Cygwin). Necesario para ejecutar comandos como *make* o *gcc*.
- Proyecto LUFA como base. Se parte del ejemplo de *firmware Joystick* para el desarrollo.
- Set de herramientas AVR, disponible en la web de Atmel (Linux o Windows), utilizado por LUFA para la compilación de los *firmwares*.

Los *firmwares* desarrollados con LUFA están estructurados en una serie de ficheros:

- Descriptors.h y Descriptors.c: Estos ficheros contienen la configuración necesaria para que el *host* sepa de qué dispositivo HID se trata y que información lo compone (conocido como descriptor HID).
- *Dispositivo*.h y *Dispositivo*.c: Son los ficheros que contienen la funcionalidad principal del dispositivo. Adaptándose al formato del descriptor HID se debe de encargar de procesar los cambios en el estado del dispositivo para notificarlos al *host*. En caso de que el dispositivo HID también recibiera datos del *host*, los datos también serían procesados en estos ficheros.
- Makefile: Contiene la información necesaria para compilar los *firmwares* (Tipo de microcontrolador, arquitectura de la placa, placa utilizada, ruta de la librería LUFA, comandos de compilación, etc.)

5.3 Desarrollo del *firmware*

Para hacer adaptar el ejemplo de *firmware* del Joystick que viene con la librería LUFA a la placa de Arduino se ha realizado una serie de cambios que se van a ir describiendo en esta sección.

5.3.1 Descriptores HID

Para indicar los descriptores HID que se van a emplear, es necesario modificar la variable *JoystickReport* del archivo Descriptors.c de modo que se adapte a nuestras necesidades. En el ejemplo original se hacía uso de la macro *HID_DESCRIPTOR_JOYSTICK* pero en este caso no se va a utilizar puesto que esta macro también indica al *host* que hay un eje Z y en el *firmware* solo se va a incluir los elementos que se usen. De este modo se crea el descriptor HID teniendo en cuenta las necesidades:

```
const USB_Descriptor_HIDReport_Datatype_t PROGMEM JoystickReport[] =
{
   HID_RI_USAGE_PAGE(8, 0x01),                  //Generic Desktop
   HID_RI_USAGE(8, 0x04),                    //Joystick
   HID_RI_COLLECTION(8, 0x01),                  //Application
      HID_RI_USAGE(8, 0x01),                 //Pointer
      HID_RI_COLLECTION(8, 0x00),               //Physical
         HID_RI_USAGE(8, 0x30),              //Eje X
         HID_RI_USAGE(8, 0x31),              //Eje Y
         HID_RI_LOGICAL_MINIMUM(16, 0),         //Valor mínimo
         HID_RI_LOGICAL_MAXIMUM(16, 1023),      //Valor máximo
         HID_RI_REPORT_SIZE(8, 16),             //Tamaño del valor de cada eje (en bits)
         HID_RI_REPORT_COUNT(8, 2),             //Nº de datos enviados (ejes)
         HID_RI_INPUT(8, HID_IOF_DATA | HID_IOF_VARIABLE | HID_IOF_ABSOLUTE),
      HID_RI_END_COLLECTION(0),
      HID_RI_USAGE_PAGE(8, 0x09),               //Button
      HID_RI_USAGE_MINIMUM(8, 0x01),            //Nº más bajo de botón
      HID_RI_USAGE_MAXIMUM(8, 6),               //Nº más alto de botón
      HID_RI_LOGICAL_MINIMUM(8, 0x00),       //Valor mínimo
      HID_RI_LOGICAL_MAXIMUM(8, 0x01),       //Valor máximo
      HID_RI_REPORT_SIZE(8, 0x01),           //Tamaño del valor de cada botón (en bits)
      HID_RI_REPORT_COUNT(8, 6),                //Nº de datos enviados (botones)
      HID_RI_INPUT(8, HID_IOF_DATA | HID_IOF_VARIABLE | HID_IOF_ABSOLUTE),
      HID_RI_REPORT_SIZE(8, 2),              //Tamaño del valor (Nº bits restantes hasta
completar byte)
      HID_RI_REPORT_COUNT(8, 0x01),          //Nº de datos (siempre 1)
      HID_RI_INPUT(8, HID_IOF_CONSTANT),
   HID_RI_END_COLLECTION(0)
}
```

Código 6: Descriptor HID del firmware para el mando de juegos

Para lograr esta configuración del descriptor es necesario consultar en la especificación HID la información que se debe introducir. Los campos relevantes están comentados en el código. En el propio fichero Descriptors.c y de forma opcional, también se pueden cambiar datos como el nombre del fabricante o el nombre del producto.

5.3.2 Envío de mensajes al *host*

Partiendo del descriptor HID, hay que modificar la estructura *USB_JoystickReport_Data_t* del archivo Joystick.h para que cumpla esta especificación.

```
typedef struct
{
   uint16_t  X; /**< Current absolute joystick X position, as an unsigned 16-bit integer */
   uint16_t  Y; /**< Current absolute joystick Y position, as an unsigned 16-bit integer */
   uint8_t Button; /**< Bit mask of the currently pressed joystick buttons */
} USB_JoystickReport_Data_t;
```

Código 7: Estructura utilizada para el envío de mensajes al host

Para enviar la información al *host* se utiliza la función de *callback* *CALLBACK_HID_Device_CreateHIDReport*. Como el propósito del *firmware* es enviar los datos que han sido obtenidos en Arduino y enviados mediante el puerto serie, se crea una variable global del tipo USB_JoystickReport_Data_t y en esta función únicamente va a enviarse el contenido de esta variable, que será actualizada con los mensajes de Arduino desde otro lugar del programa.

```
USB_JoystickReport_Data_t reportJoystick = {0, 0, 0};
…
bool CALLBACK_HID_Device_CreateHIDReport(USB_ClassInfo_HID_Device_t* const HIDInterfaceInfo,
                                         uint8_t* const ReportID,
                                         const uint8_t ReportType,
                                         void* ReportData,
                                         uint16_t* const ReportSize)
{
   USB_JoystickReport_Data_t* report = (USB_JoystickReport_Data_t*)ReportData;

   *report = reportJoystick;

   *ReportSize = sizeof(USB_JoystickReport_Data_t);

   return false;
}
```

Código 8: Código de la función CALLBACK_HID_Device_CreateHIDReport

5.3.3 Implementación del protocolo de mensajes

Para implementar el protocolo de mensajes mediante el puerto serie es necesario primero inicializarlo en la función *SetupHardware* mediante la instrucción *Serial_Init(9600, true);* para luego leer los mensajes en el bucle principal del programa. Para leer un byte, se ha creado una función llamada *getNewSeriaByte* que, además de recibir un byte del puerto serie, ilumina el led Tx de la placa para indicar que se están transmitiendo datos.

```
#define LED_ON_TICKS 2000

int main(void)
{
   uint8_t serialByte1, serialByte2, serialByte3;
   SetupHardware();

   GlobalInterruptEnable();

   for (;;)
   {
      HID_Device_USBTask(&Joystick_HID_Interface);
      USB_USBTask();

      if(Serial_IsCharReceived())
      {
         serialByte1 = getNewSerialByte();

         if (serialByte1 != 'B' && serialByte1 != 'X' && serialByte1 != 'Y')
            continue;

         // Necesitamos al menos un byte mas
         while (!Serial_IsCharReceived())
            ;
         serialByte2 = getNewSerialByte();


         if (serialByte1 == 'B')
         {
            uint8_t btnNumber = (serialByte2 >> 4);
            uint8_t value = serialByte2 & 0x1;

            if (value)
               reportJoystick.Button |= value << (btnNumber-1);
```

```
            else
                reportJoystick.Button &= ~(~value << (btnNumber-1));
        }
        else if (serialByte1 == 'X' || serialByte1 == 'Y')
        {
            while (!Serial_IsCharReceived())
                ;
            serialByte3 = getNewSerialByte();
            uint16_t value = (serialByte2 << 8) + serialByte3;
            if (serialByte1 == 'X')
                reportJoystick.X = value;
            else
                reportJoystick.Y = value;
        }
      }
    }
}


uint8_t getNewSerialByte(void)
{
    LEDs_TurnOnLEDs(LEDMASK_TX);
    ledTicks = LED_ON_TICKS;
    for (;;)
    {
        ledTicks--;
        if (!ledTicks)
            break;
    }
    LEDs_TurnOffLEDs(LEDMASK_TX);

    return (uint8_t) Serial_ReceiveByte();
}
```

Código 9: Implementación del protocolo de mensajes a través del puerto serie

Para terminar esta parte sólo queda compilar con el comando *make* y *flashear* el archivo .hex resultante en el microcontrolador ATMega16u2 a través del *software* Flip, pero antes hay que adaptar y *flashear* en Arduino el *firmware* del mando para juegos.

5.3.4 Adaptación del protocolo al *software* del mando para juegos

El hecho de diseñar un *firmware* que se ajusta perfectamente a las características del dispositivo se puede eliminar partes de código innecesario. De forma general se puede decir que ya no se envía la misma estructura con el estado de los ejes y los botones en cada iteración (se envía únicamente los cambios), ni es necesario utilizar funciones como map() para ajustar los valores leídos de los ejes (ahora ya se le indica al *host* mediante el descriptor HID que se le enviará un valor entre 0 y 1023). Debido a que sólo se envían los cambios, hay una espera de 3 segundos tras de iniciar el puerto serie para dar tiempo a que también se inicie en la parte del *firmware* y no se pierdan datos. Tras estos 3 segundos se envía el estado inicial de cada uno de los botones y ejes.

```
void loop() {
   for (int i = 0; i < numButtons; i++) {
      int val = digitalRead(inputButtons[i]) == HIGH ? 1 : 0;
      if (val != lastBtnValues[i]) {
         reportBtnValue(i, val);
         lastBtnValues[i] = val;
      }
   }

   int xAxis = analogRead(X_PIN);
   if (xAxis != lastAxisValues[0])
   {
      int diff = xAxis - lastAxisValues[0];
      if (abs(diff) > 1)
      {
         reportAxisValue('X', xAxis);
         lastAxisValues[0] = xAxis;
      }
   }

   int yAxis = analogRead(Y_PIN);
   if (yAxis != lastAxisValues[1])
   {
      int diff = yAxis - lastAxisValues[1];
      if (abs(diff) > 1)
      {
         reportAxisValue('Y', yAxis);
         lastAxisValues[1] = yAxis;
      }
   }
   delay(50);
}
```

Código 10: Bucle principal del programa del mando para juegos, adaptado al protocolo del nuevo firmware

A modo de detalle, se ha observado que el sensor del joystick reportaba variaciones en la lectura de 1 unidad mientras se mantenían los ejes parados, de modo que no se reporta un cambio a no ser que la lectura varíe en más de una unidad respecto a la anterior lectura.

6. Conclusiones

Como se ha podido observar a lo largo de la memoria, se ha logrado el objetivo principal, que no era otro que diseñar un dispositivo HID basado en un microcontrolador. Se ha optado por utilizar una plataforma libre tanto *software* como *hardware* para disponer de mayor flexibilidad y un coste menor. Para lograr este objetivo se ha utilizado la plataforma Arduino y se ha contado con la ayuda de librerías de terceros, tanto para controlar sensores y actuadores de Arduino como para desarrollar el *firmware* HID (librería LUFA).

Al lograr el objetivo principal también se han logrado los secundarios, es decir, se ha aprendido a programar los sensores y actuadores en Arduino y a conectarlos físicamente a éste, se ha aprendido el funcionamiento del protocolo HID y se ha conseguido crear un *firmware* para hacer realizar la comunicación entre Arduino y el *host* a través de un puerto USB con controlador HID.

De este modo, queda demostrado que se puede construir un dispositivo HID con las características deseadas, poniendo como límite la imaginación, los sensores existentes y el propio protocolo HID.

Como trabajo futuro se podría desarrollar un *firmware* para el resto de los dispositivos que se han implementado en la memoria y en los que se ha utilizado un *firmware* creado por una tercera persona. Otra opción sería crear por completo (*software*, *hardware* y *firmware*) un dispositivo específico para cubrir una necesidad especial de alguna persona, ya sea por cuestiones de discapacidad física / psicológica como por comodidad o ergonomía.

7. Referencias

A continuación se enumeran las fuentes consultadas y la información obtenida en ellas:

- https://es.wikipedia.org Para consultar definiciones y otra información básica.
- https://www.arduino.cc/ Web de Arduino para consultar la documentación relativa a las funciones, para descargar librerías y para consultar las características de la placa utilizada (Arduino Mega 2560).
- http://www.fourwalledcubicle.com/LUFA.php Web de la librería LUFA, utilizada para construir el *firmware*. Descarga de la librería y documentación online.
- http://hunt.net.nz/users/darran/ Web del creador de los *firmwares* utilizados en los dispositivos del Ratón, Teclado y Mando para juegos. Descarga de los *firmwares* y consulta de ejemplos.
- https://github.com/ddiakopoulos/hiduino GitHub del proyecto HIDUINO. De esta web se ha obtenido el *firmware* utilizado en el teclado MIDI.
- http://www.usb.org/developers/hidpage/HID1_11.pdf Especificación HID, útil para crear el descriptor HID a incluir en el *firmware*.
- http://eleccelerator.com/tutorial-about-usb-hid-report-descriptors/ Tutorial de creación de descriptores HID