LEVITADOR ACÚSTICO MULTIEMISOR NO RESONANTE DE EJE ÚNICO

Mantenimiento electrónico S21ME

Autor: Christian Álvarez Folkmann

Tutor: Enrique Norro

Curso: 2019/2020

ÍNDICE

1. Resumen 3
2. Justificación del proyecto 3
3. Objetivos 4
3.1 Objetivo general 4
3.2 Objetivos específicos 4
4. Desarrollo 4
4.1 Revisión bibliográfica, fundamentación teórica 4
4.2 Materiales y métodos 7
4.3 Resultados y análisis 12
4.3.1 Montaje 13
4.3.2 Programación 21
5. Conclusiones 39
6. Líneas de investigación futuras 39
7. Temporalización 40
8. Bibliografía 41
9. Webgrafía 42

1. Resumen

El presente proyecto tiene por objeto la construcción y explicación del funcionamiento de un levitador no resonante de eje único con componentes electrónicos de bajo coste. Se demuestra la levitación de objetos de pequeña masa con un consumo de entrada bastante pequeño.

Un levitador acústico es un dispositivo capaz de mantener unas partículas y elementos en suspensión en un rango de alturas determinado. Los elementos pueden ser ferromagnéticos, paramagnéticos y diamagnéticos, por lo que no tiene limitaciones en cuanto a estas propiedades.

La levitación acústica está siendo muy estudiada a nivel mundial debido a las grandes ventajas que puede aportarnos. Es un campo de investigación emergente que incorpora en su estudio numerosas herramientas del entorno de la electrónica.

Con este trabajo se conocerán todos los antecedentes que han llevado al nivel actual y se entenderán todos los conceptos físicos que hay detrás de este fenómeno, al igual que se profundizará en la electrónica y programación necesaria para realizarlo.

Gracias a un proyecto de investigación internacional liderado por el ingeniero español Asier Marzo Pérez se realizó un modelo en pequeña escala de los levitadores usados en sus laboratorios, creando así un proyecto de código libre, el cual he utilizado como base para realizar este proyecto.

Es aquí donde entra mi levitador, que se trata de un levitador no resonante de eje único capaz de producir un atrapamiento estable con un bajo voltaje, funcionando de manera consistente incluso con cambios de temperaturas y por períodos prolongados de tiempo. Usamos 72 transductores para las matrices en vez de bocinas de Langevin convencionales.

Tendremos que realizar un buen análisis de contenido bibliográfico, así como observación y una buena estructura de la programación y diseño electrónico que nos permitan seguir un procedimiento que garantice finalizar el proyecto con éxito.

2. Justificación del proyecto

Lo que me ha llevado a realizar este proyecto es mi interés por la demostración de conceptos físicos fundamentales a través de la electrónica. Mi motivación es el estudio de experimentos científicos de mucho coste económico y ser capaz de reproducirlos de manera económica gracias a la gran cantidad de componentes electrónicos disponibles en el mercado.

Este proyecto en concreto puede usarse como base para futuros avances en una gran cantidad de campos científicos en los que se puede aplicar. La versatilidad de la levitación acústica la convierte en una herramienta útil dentro de la química farmacéutica, medicina, espectroscopia, estudio de dinámica de fluidos y tensión superficial, levitación de muestras biológicas, estudio de transiciones de fase, rápida cristalización o ionización de muestras, etc.

También es muy llamativo visualmente, lo que anima bastante su elaboración e investigación.

3. Objetivos

3.1 Objetivo general

- Construir un levitador acústico multiemisor no resonante de eje único.

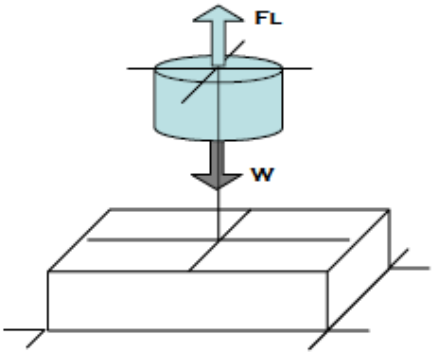
3.2 Objetivos específicos

- Definir la física que interviene en el proceso de levitación acústica.
- Determinar las características y el funcionamiento de los transductores.
- Realizar una estructura estable que contendrá las matrices de transductores.
- Distinguir las partes electrónicas que intervienen en el proyecto.
- Conseguir una programación que se adecúe a los objetivos físicos del proyecto.

4. Desarrollo

4.1 Revisión bibliográfica, fundamentación teórica

Lo primero que se debe entender es el concepto de levitación. Desde el punto de vista físico para que se presente la levitación es necesario una fuerza que contrarreste el peso del cuerpo para que se mantenga en suspensión estable. Este fenómeno ha de darse ante la presencia de un campo gravitatorio.



Dentro de la levitación existen varios tipos: Electrostática, magnética, aerodinámica, óptica, acústica, antigravedad (teórica). De todas estas la levitación acústica es la que mejores propiedades tiene para nuestros objetivos, debido a que el sonido es una onda mecánica y, como tal, cuenta con una energía que al incidir en un objeto puede ser atrapado y levitado de manera estable.

El principio de un levitador acústico es la generación de una onda estacionaria desde un emisor, llamado transductor, hacia un reflector cóncavo separados por una distancia múltiplo entero de media longitud de onda. Esto produce reflexiones e interacción entre todas las ondas generadas, creando nodos y vientres en los que hay una diferencia de presión. En estos nodos de presión se incrementa la amplitud y la partícula situada en ese punto levitará alcanzando el equilibrio estático.

Los levitadores más comunes son los de eje único, entre los cuales se distinguen dos tipos. El primero cuenta como hemos dicho anteriormente con un transductor acústico y un reflector que actúa como cavidad resonante, por ese motivo a este se le llama levitador resonante. Por otro lado, también podemos realizar un levitador no resonante con dos emisores opuestos que producen ondas estacionarias dando lugar a las interacciones entre estas.

El pequeño inconveniente que se puede encontrar es que un cambio de temperatura varía la velocidad del sonido, afectando directamente a la longitud de onda y produce una reducción de la fuerza de captura. (Xie & Wei, 2003).

Los investigadores suelen usar los levitadores no resonantes debido a que son bastante estables, versátiles y dan buenos resultados para temperaturas entre - 40 y 40 ºC.

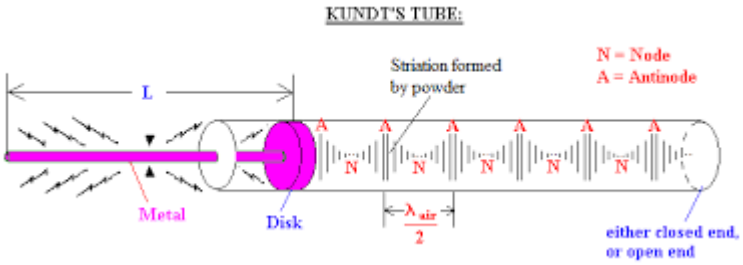
Se han realizado estudios sobre la eficiencia y la fuerza de atrapamiento de las partículas (Oran, Berge, & Parker, 1980) y se llegó a la conclusión de que una disposición cóncava de los transductores producía una fuerza mayor que la disposición plana, debido a la concentración de energía acústica en un punto concreto.

Los resultados de este estudio se validaron posteriormente. (Xie & Wei, 2001) y (Xie & Wei, 2002).

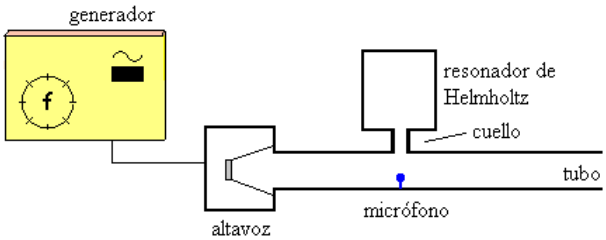
La acústica es una ciencia en la que se ha comenzado a profundizar hace relativamente poco tiempo, con lo cual los antecedentes del proyecto son bastante recientes.

El primer informe técnico que se publicó fue de la NASA en 1975 donde se explica brevemente el funcionamiento de un levitador de partículas que cuenta con un emisor de sonido y un reflector acústico. (Whymark, 1975)

Los primeros trabajos experimentales sobre levitación acústica se remontan a 1985, cuando W. Jackson, M. Barmatz y P. Wagner realizan un sistema de levitación usando un solo transductor en un tubo de Kundt, el cual fue ideado en un principio para el estudio de ondas estacionarias por August Kunt. (Jackson, Barmatz, & Wagner, 1994)



Dos años más tarde, en 1987, David Deak comienza a hacer estudios de micro gravedad para la NASA. Para ello construye una cámara de levitación con 3 altavoces basada en la cavidad resonante de Helmholtz. Fue la primera vez que se conseguía controlar el movimiento de las partículas en suspensión en los tres ejes (x,y,z).



En 1992 Wenjun Xie de Northwestern Polytechnical University (China) comenzó a emplear un emisor y un reflector que producen los fenómenos explicados anteriormente. Con este método consiguió una longitud de onda de 20 mm, por lo que podría levitar objetos de esas dimensiones. Tenía una curiosidad, ¿qué pasará si un animal vivo es puesto en el campo acústico? Por eso se centró en los efectos que tendría sobre animales como arañas, hormigas o incluso peces de pequeño tamaño. El resultado fue que los seres vivos no sufren efectos negativos al verse atrapados por estos campos acústicos. Todos los resultados y explicación detallada se encuentran en artículos más recientes (Xie W. J., Wei, Cao, Lü, & Hong, 2006).

Posteriormente, en 1996 el físico M. Stieglmeier desarrolla un levitador para la ESA (European Space Agency) y comienzan a realizar experimentos que antes resultaban imposibles o demasiado costosos.

Gregory Mc Daniel y Glynn Holt desarrollan en 1998 una técnica experimental para el Glenn Research Center de la NASA con unos resultados bastante satisfactorios y un avance en las demostraciones acústicas. Publicaciones realizadas: (Gregory & Glynn, 2000)

Más recientemente en 2008 GEA Process Engineering situado en Dinamarca saca al mercado su Drying Kinetics Analyzer, el cual se comienza a usar para el análisis de dinámica de fluidos. Consta de un emisor que genera una onda sinusoidal de 58 KHz, y un reflector cóncavo a una distancia de 5/2 λ. También es capaz de mantener la temperatura deseada para evitar desajustes de resonancia.

Los científicos Yoichi Ochiai, Takayuki Hoshi y Jun Rekimoto de la universidad de Tokio, realizan un estudio de investigación que publican en 2014 *Three-Dimensional Mid-Air Acoustic Manipulation by Ultrasonic Phased Arrays.* Traducido al español se conoce como "Manipulación acústica tridimensional en el aire mediante matrices por fases ultrasónicas". (Ochiai, Hoshi, & Rekimoto, 2014)

Se consigue un gran cambio en la cultura científica proponiendo un novedoso sistema de gráficos basado en la expansión de la tecnología de manipulación acústica 3D. En vez de manipular partículas en un eje fijo mediante la diferencia de fases de los transductores de tipo Langevin, se manipulan mediante matrices de transductores que generan ondas estacionarias localizadas permitiendo el control tridimensional e individual de cada partícula.

El método propuesto en este artículo permite trabajar con partículas de poliestireno expandido de 0.6 mm y 2 mm de diámetro. Cambiando la distribución de campo de potencial acústico se consigue una resolución espacial de la posición de 0,5 mm.

En 2015 el ingeniero español Asier Marzo Pérez de la Universidad Pública de Navarra en colaboración con investigadores de la universidad de Bristol y Sussex ha conseguido impulsar los estudios sobre este tema. Lo que ha desarrollado en su tesis *Holographic acoustic elements for manipulating levitating particles* "Hologramas acústicos para la levitación de partículas" es la elaboración de un rayo tractor sónico que consigue atrapar y atraer la partícula hacia la fuente emisora del sonido (Marzo Pérez, Holographic Acoustic Elements for Manipulating Levitating Particles, 2015). En todos los casos anteriores vimos como se trataba de una fuente emisora que empujaba una partícula con las ondas del sonido y mediante resonancia o interferencias de las mismas levitaba, pero en este caso es la primera vez que se consigue hacer al contrario atrayéndola hacia la fuente. De esta manera se consigue una levitación mucho más llamativa y práctica a la hora de ser usado en otros campos.

También consiguieron en el 2018 mediante un rayo tractor más potente levitar el mayor objeto con ultrasonido hasta el momento, se trataba de una partícula de 2 cm de diámetro.

4.2 Materiales y métodos

Metodología de Diseño Electrónico

La metodología de diseño debe garantizar que se minimicen los errores, y que, caso de que se produzcan, éstos se detecten en la fase más temprana posible.

Describiremos las consideraciones de diseño para hacer un levitador de un solo eje utilizando una matriz de pequeños emisores ultrasónicos. Primero, analizamos los transductores disponibles que son los elementos que transforman la señal eléctrica en potencia acústica. En segundo lugar, estudiamos cómo organizar espacialmente los transductores para maximizar las fuerzas de captura y el número de trampas, así como para reducir los reflejos parásitos. Tercero, presentamos electrónica simple y efectiva para manejar los transductores.

Técnicas de Análisis de Contenido

Análisis de la bibliografía, así como del proceso evolutivo en el contexto de la física. Proporcionando la revisión de los antecedentes de los proyectos realizados, analizando sus contenidos tanto históricos como actuales. Al mismo tiempo que el estudio dentro del contexto nacional e internacional de proyectos similares, de levitadores ligados al mundo de la electrónica y en especial del proyecto de código libre de Asier Marzo.

Técnicas de Observación

Realizamos la observación de los hechos, fenómenos y otros estudios con el fin de determinar la información necesaria para realizar el proyecto.

Procedimiento

Después de haber planteado las bases teóricas, se procedió a la aplicación de dichas técnicas de forma contextual al proyecto a realizar, se comenzó por señalar los pasos a seguir en cada una de las técnicas, de cómo se va a hacer, así como el planteamiento de los materiales e instrumentos de medición.

Materiales

- x 72 Transductor MCUSR10P40B07RO

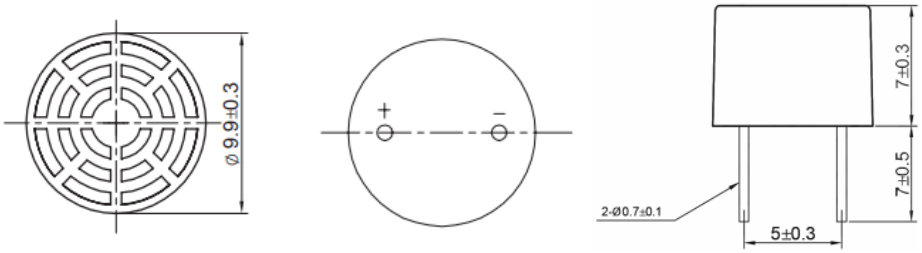
Estos son los componentes fundamentales del levitador. Son los encargados de transformar la señal eléctrica de entrada en señal acústica a la salida.

Frecuencia: 40 KHz

Directividad: 50º

Temperatura de trabajo: -35 a 85 ºC

Diámetro: 10 mm



- x 1 Arduino Nano

Las características del Arduino Nano son más que suficientes para este proyecto. Esto junto con su reducido tamaño favorece para hacer un proyecto eficiente y más compacto.

Microcontrolador: Atmel ATmega328p

Voltaje de trabajo: 5 V

Voltaje de entrada: 7-12V

Frecuencia: 16 MHz

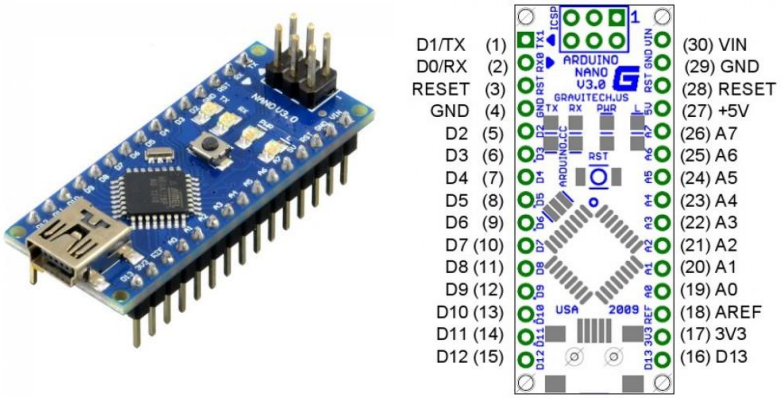
Nº de entradas analógicas: 8

Nº de pines digitales: 14

Nº pines PWM: 6

Intensidad máxima en pin: 40 mA

Dimensiones: 48 x 18 mm



- x 1 Módulo Driver L298N

Este módulo está construido en torno al circuito integrado L298N, el cual dispone en su interior de dos puentes en H encargados de amplificar la señal que controla los transductores.

Voltaje de entrada: 5-12 V

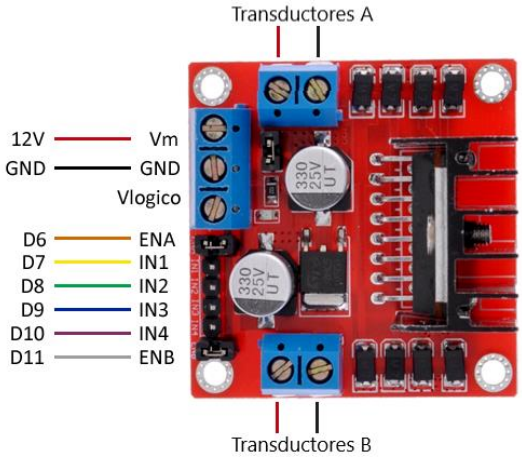
Consumo de corriente: 0 a 36 mA

Capacidad de corriente: 2 A

Potencia máxima: 25 W

Canales: 2

Dimensiones: 43 x 43 x 27 mm



- <u>x 1 Adaptador AC/DC</u>

INPUT: 100-240VAC 50/60 Hz

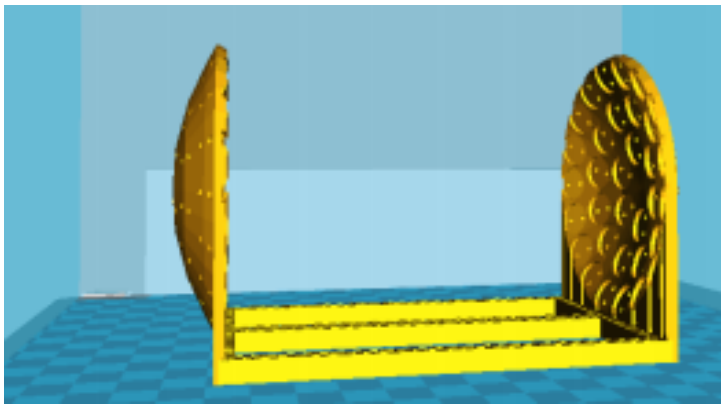
OUPUT: 9VDC 1 A

- <u>Conector DC</u>

- <u>x 3 Pulsador</u>

- <u>x 1 Jumper</u>

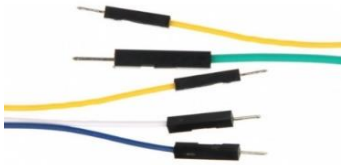
- x 1 Estructura impresa 3D



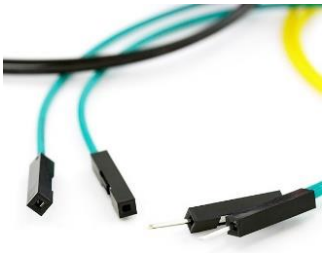
- Láminas de aluminio

- Cable rojo y negro 12 AWG

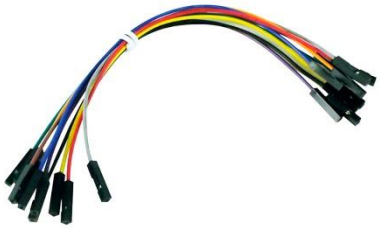
- Cables macho/macho



- Cables macho/hembra



- Cables hembra/hembra



Herramientas necesarias

- Impresora 3D (Se puede usar un servicio de impresión 3D online)
- Polímetro
- Soldador
- Taladro
- Destornilladores
- Alicates
- Pegamento instantáneo
- Osciloscopio* (No es obligatorio, pero ayuda para los test de transductores)

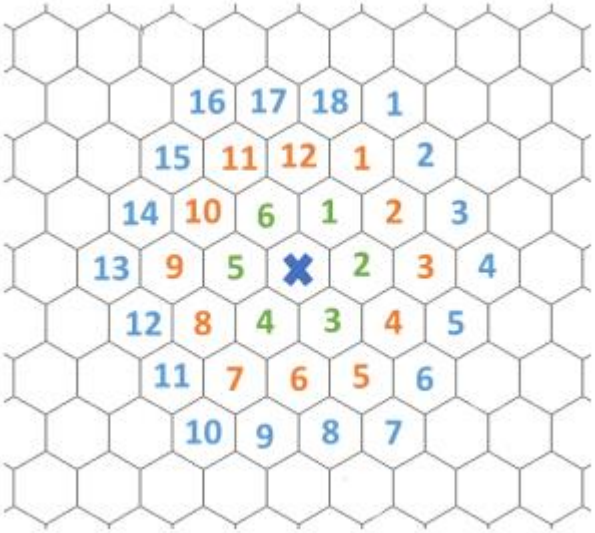
Coste del proyecto

Materiales	Coste
Transductor MCUSR10P40B07RO	0,32 € / pza = 23,68 €
Arduino Nano	3,79 € + 0,99 € envío = 4,78 €
Driver L298N	2,72 €
Adaptador AC/DC	3,40 € + 1,40 € envío = 4,80 €
Conector DC	0,37 €
Pulsador	0,10 € / pza = 0,3 €
Jumper	0,02 €
Estructura 3D	2 €
Lámina de aluminio	7 €
Cables	4 €
TOTAL	**49,67 €**

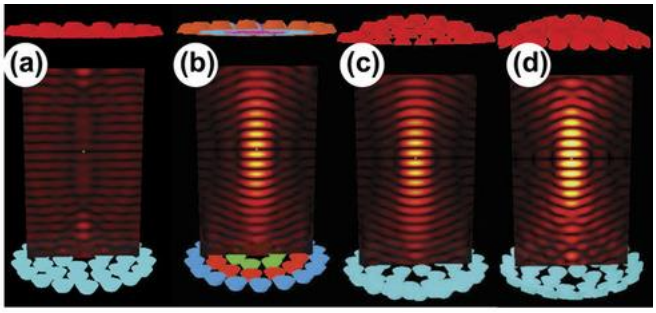
También existe la posibilidad de adquirir algún kit que contenga todos los materiales. Se puede conseguir por 70 € más gastos de envío, lo que se quedaría en 75,58€. De esta manera sale más caro pero te ahorras buscar todas las piezas de manera individual por distintas páginas de compra online.

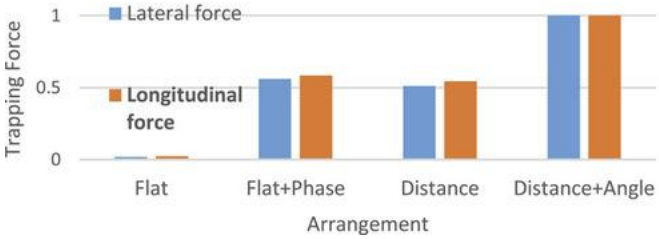
4.3 Resultados y análisis

Tras el estudio realizado de los aspectos físicos y teóricos se comienza con el montaje del levitador. Como se puede observar en el apartado de materiales, los transductores son los componentes principales, contando con 36 emisores en cada matriz (72 en total). Estos están montados formando tres anillos de 6, 12 y 18 emisores, como se muestra en el patrón hexagonal.



Se estudiaron cuatro tipos de enfoques para determinar cual tenía una mayor fuerza de captura, llegando a los siguientes resultados.





a) Sin enfoque. Muestra la disposición plana de los transductores, la cual da una fuerza de captura demasiada baja para los objetivos de nuestro proyecto.

b) Enfoque de fase electrónica. Contamos con una disposición plana aunque en este caso se ajusta electrónicamente la fase de las señales, consiguiendo una mejora considerable. Este enfoque no se siguió en el trabajo debido a la dificultad electrónica que conlleva generar muchas señales independientes.

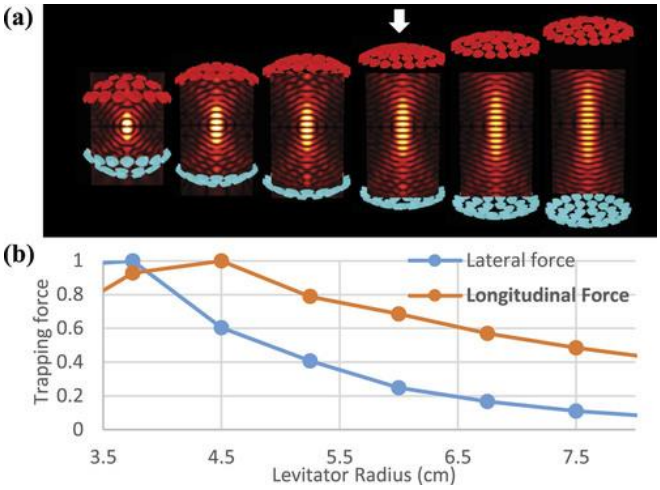
c) Enfoque por desplazamiento de distancia. Es posible crear los mismos desfases y conseguir el mismo rendimiento que en el caso anterior variando las distancias verticales de los transductores.

d) Enfoque por desplazamiento de distancia y ángulo de orientación. Se parte del desplazamiento de distancia anterior, sumándole una orientación de los transductores hacia el foco, donde conseguiremos una intensidad máxima.

Por estos buenos resultados del enfoque **d** se eligió como mejor opción para el proyecto.

Una vez tenemos claro el tipo de enfoque que se ve a utilizar también hay que tener en cuenta la separación entre ambas matrices de transductores. En la siguiente imagen se muestra como varía la fuerza generada dependiendo de la distancia de separación.

Es importante señalar que cuando aumenta la distancia debemos aumentar también el radio de orientación de los transductores para que se dirijan siempre al foco.

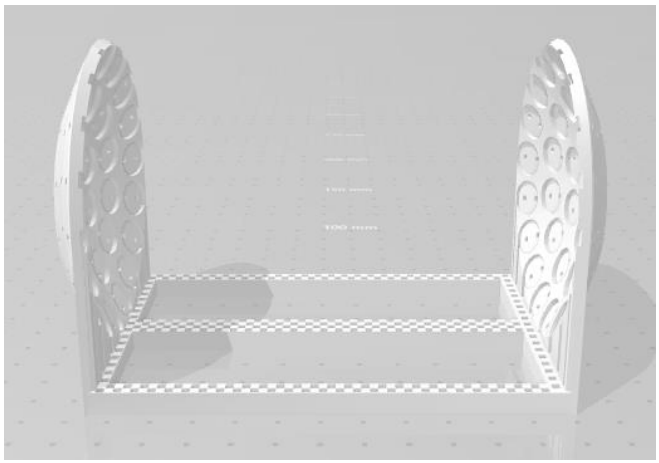


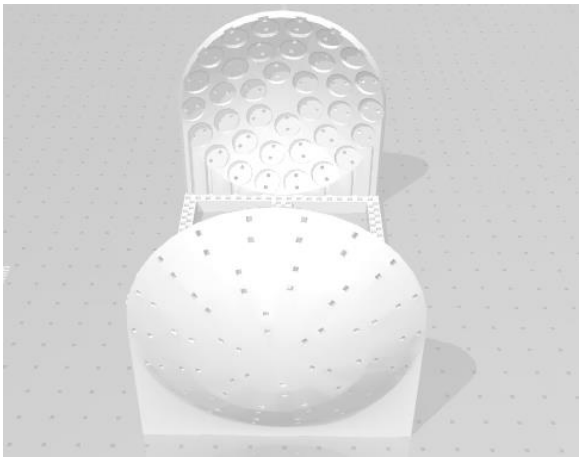
Cuando nos fijamos en la gráfica lo más importante es la “Longitudinal Force”, que es la que produce la levitación. En cambio, la “Lateral Force” puede ser menor ya que solo se encarga de la estabilidad lateral.

Se puede observar en la gráfica que la fuerza máxima se consigue con un radio de 4,5 cm, aunque seleccionamos el radio de 6 cm para evitar interferencias y reflejos no deseados.

4.3.1 Montaje

Una vez tenemos todos los datos y medidas podemos realizar el montaje de la estructura sobre la que se colocarán las matrices de transductores. Para ello contamos con un diseño realizado en 3D que nos da una estructura ligera y compacta.





Los transductores que hemos elegido funcionan a 40 KHz, lo que nos da una longitud de onda de 8,5 mm en el aire a 20 ºC. Como hemos visto en la parte teórica, podremos levitar un objeto que mida la mitad de la longitud de onda, en este caso es de aproximadamente 4 mm.

$$\lambda = \frac{Vs}{f} = \frac{340\ m/s}{40\ KHz} = 8{,}5\ mm$$

Donde;

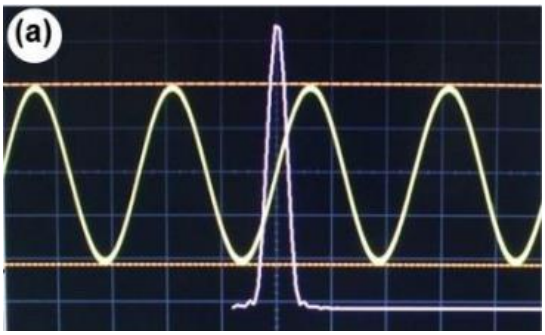
λ = Longitud de onda

Vs = Velocidad del sonido en el aire a 20 ºC

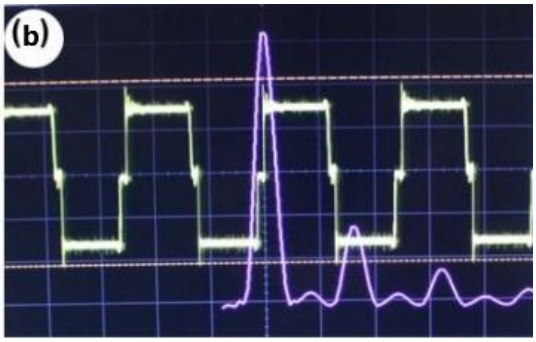
f = Frecuencia de los transductores

Se estudió la posibilidad de usar otros transductores como los Manorshi MSO-P1640H10TR, MSO-P1040H07T o los MCUSD16A40S12RO. Aunque finalmente hemos elegido los MCUSR10P40B07RO debido a que son de un tamaño muy reducido, un bajo precio y unas características técnicas favorables para nuestros objetivos.

Sabemos que las ondas que hacen levitar las partículas son sinusoidales, el problema es que esas ondas son bastante complejas de generar digitalmente. La solución a la que hemos llegado es generar simplemente ondas cuadradas utilizando un Arduino Nano y visualizar en el osciloscopio cual es el resultado.



En esta imagen se puede ver una señal de excitación de una onda sinusoidal pura generada por un generador de señal.



Aquí podemos observar la señal de excitación cuadrada generada con el Arduino Nano y amplificada por el driver.

En las imágenes a y b se observan dos señales, una amarilla y otra morada. La señal amarilla es la señal en el dominio del tiempo, donde se puede observar una clara diferencia entre la sinusoidal pura y la cuadrada. La línea morada muestra la señal en el dominio de la frecuencia.

A continuación se observará en la imagen c la señal de salida del transductor, que es la que realmente nos interesa.

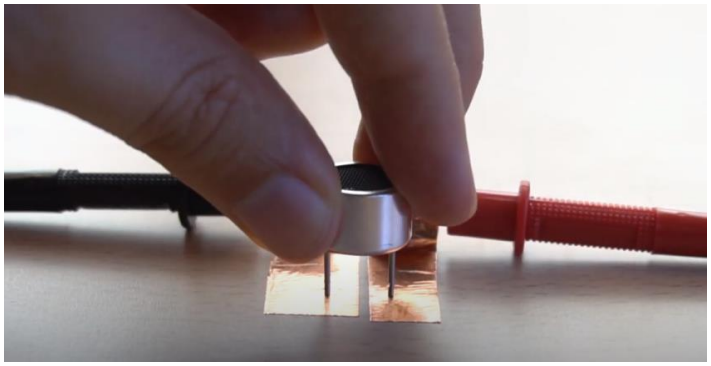
La salida del transductor es una señal sinusoidal debido a que los transductores actúan como filtro. Por ese motivo la señal de excitación que utilizamos es cuadrada, porque la salida acaba siendo muy similar a si usásemos una señal sinusoidal.

Con lo cual, la decisión final a la que hemos llegado es que generaremos señales cuadradas con un Arduino Nano que serán amplificadas con el Driver L298N.

Ahora llega el momento de montar los transductores en la estructura 3D. Para eso es muy importante tener en cuenta la polaridad de los mismos, la cual en numerosas ocasiones no coincide con lo marcado por el fabricante, debido a que se adquirieron en AliExpress a un precio muy bajo. Por ello los vamos a comprobarlos individualmente de la siguiente manera:

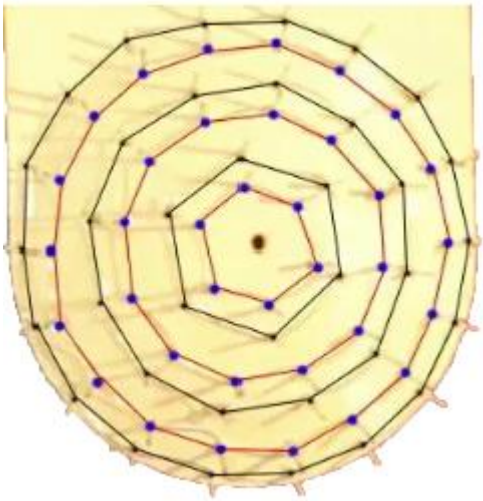
- Primero cogemos un multímetro y lo colocamos en medición de voltaje DC a la máxima sensibilidad que nos permita.

- Después medimos el transductor y si nos da un valor negativo marcaremos la patilla conectada a la punta de prueba negra, si nos diese positivo marcaríamos la patilla conectada a la punta de prueba roja.



Así estamos marcando la polaridad de los transductores, y a la hora de pegarlos a la base se colocan con la patilla marcada hacia el centro.

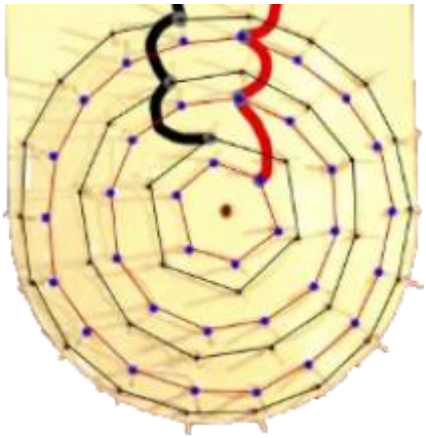
Una vez los tengamos todos montados en la estructura, necesitaremos varios cables de cobre expuestos con los que vamos a unir las patillas en seis anillos concéntricos, tal y como se muestra en la imagen.



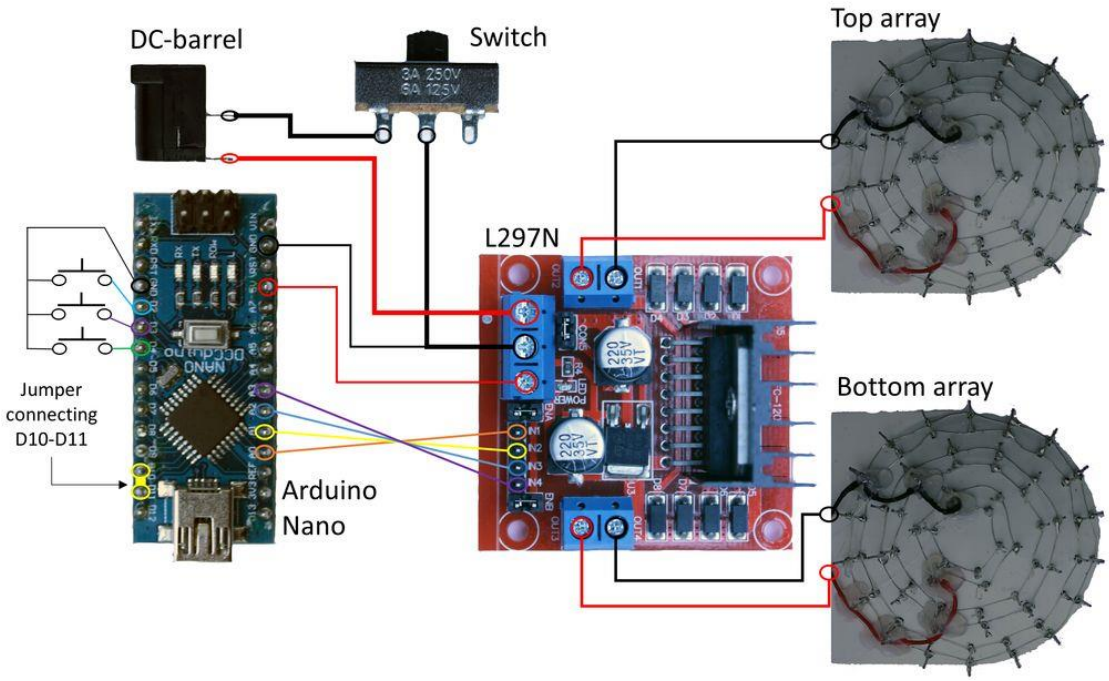
Para evitar que se suelten hemos de realizar una soldadura en cada patilla.

A continuación, con un cable rojo unimos los tres anillos correspondientes a las patillas positivas de los transductores, y con un cable negro unimos los anillos negativos.

Esto se realizará en ambas matrices de transductores, tanto la superior como la inferior.



El circuito electrónico que usamos para este proyecto se ha intentado hacer de tal manera que sea fácil identificar posibles errores y preferiblemente evitarlos tal y como explicamos en la metodología del diseño electrónico. A continuación la imagen muestra el esquema.

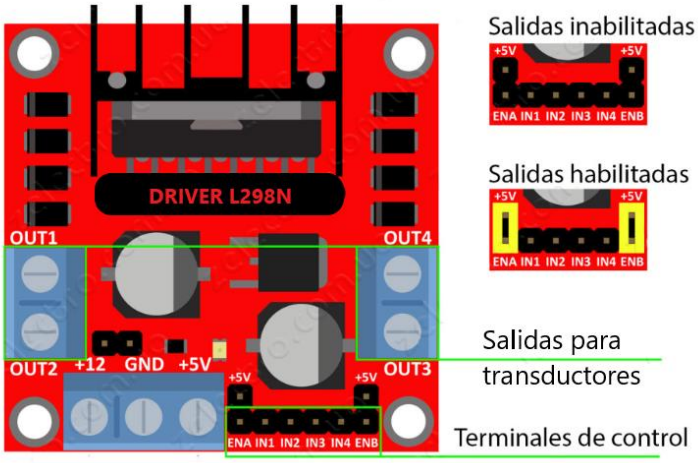


Como se puede ver, ambos lados de los transductores (Top y Bottom) están conectados al driver mediante los cables que soldamos en el paso anterior. De esta manera controlamos dos canales, uno de ellos lo mantenemos a una fase constante, mientras que el otro lo podemos variar para mover partículas en el eje longitudinal.

El driver L298N cuenta con cuatro terminales de control que vamos a conectar al Arduino. Pero antes de seguir estudiando el circuito, vamos a comprender un poco mejor la estructura del propio driver.

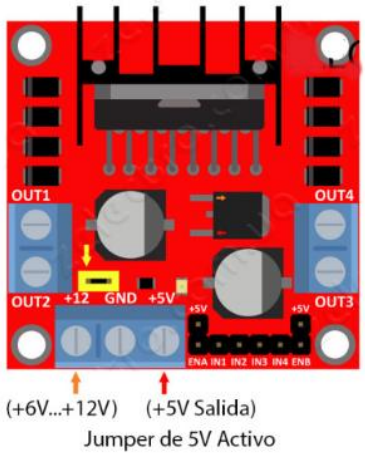
Cuenta con dos jumper de selección (ENA y ENB), mediante los cuales podemos habilitar o deshabilitar las salidas A y B. La salida A está compuesta por OUT1 y OUT2 y la salida B por OUT3 y OUT4. A través de estas se hará llegar la señal a los transductores.

Los pines de control se encuentran en la parte inferior marcados como IN1, IN2, IN3 e IN4, que se conectan mediante cables hembra a los pines del Arduino A0, A1, A2 y A3 respectivamente.



Para alimentar el driver vamos a usar un adaptador AC/DC de 9V de salida.

En la siguiente imagen se puede observar en amarillo un jumper (jumper de selección), cuando se encuentra activo permite una alimentación entre 6 y 12 V, si quitamos ese jumper los voltajes de trabajo estarían entre 12 y 35 V. En nuestro caso lo mantendremos activo ya que vamos a alimentarlo con 9 voltios, que van directamente a la entrada marcada como +12.



Como se puede observar en el esquema general del circuito (mostrado anteriormente), la salida 5V se conecta al pin 5V del Arduino, esto se utiliza para alimentar el microcontrolador.

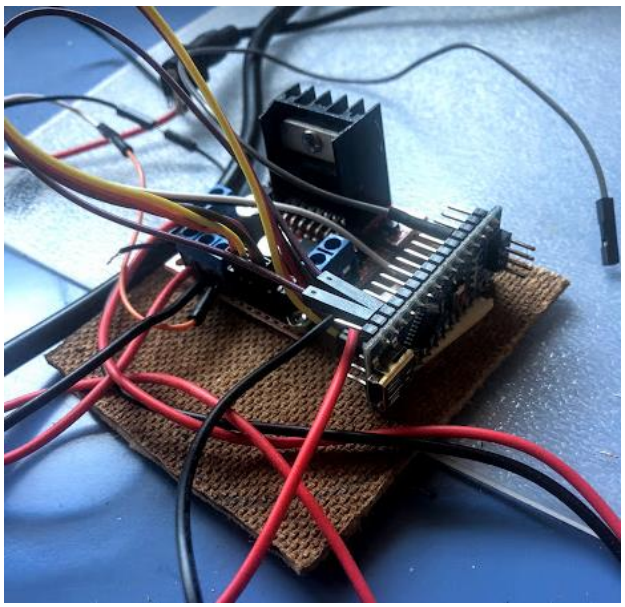
El GND es común tanto para el L298N como para el Arduino. De ahí también se conecta un cable al interruptor para poder encender y apagar sin necesidad de desconectar de la corriente.

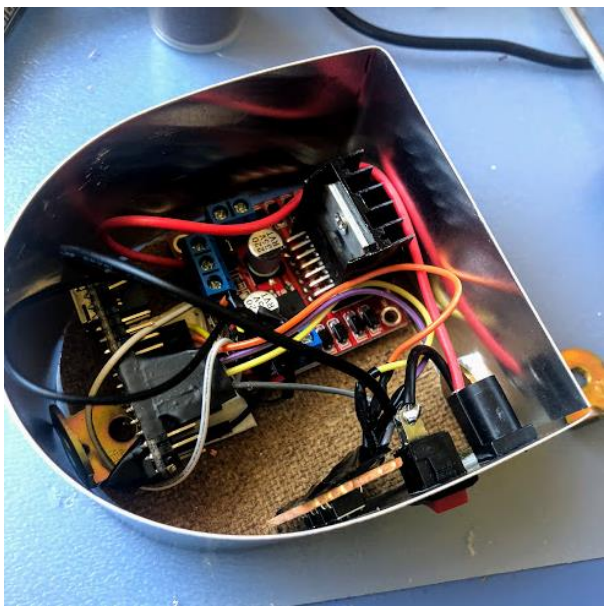
Estas son todas las conexiones que se realizan en el driver.

En el Arduino es fundamental conectar el pin D10 y D11 con un jumper para el sincronismo de la emisión de señales.

Conectaremos un pulsador en cada uno de los pines D2, D3 y D4, llevados a tierra. De esta manera podremos controlar el movimiento hacia arriba, abajo o reseteo de la partícula.

Ya con toda la electrónica montada y conectada correctamente, pasaremos a pegarla en una base que rodearemos con una lámina de aluminio para darle un acabado más estético y profesional. En esta lámina se han hecho agujeros para los pulsadores y el interruptor general.

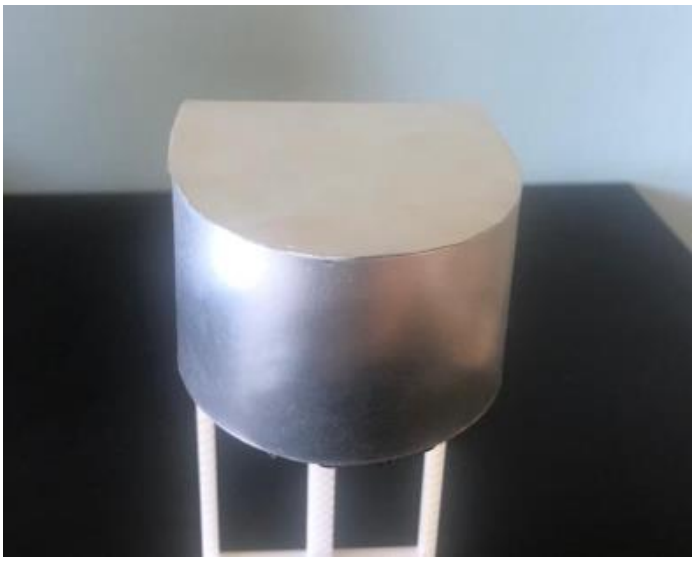




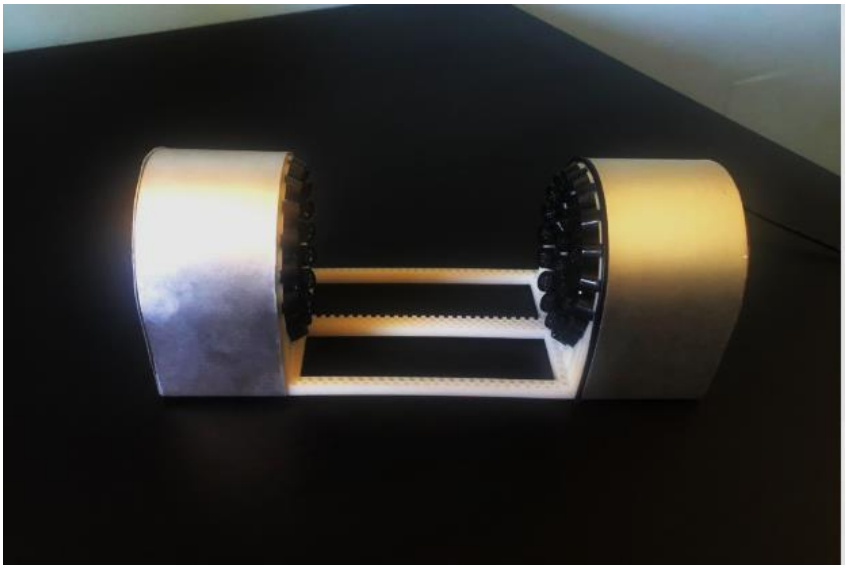
Podemos observar que se ha buscado realizar un montaje bastante compacto para que sea fácil de manipular cuando lo tengamos completamente acabado.

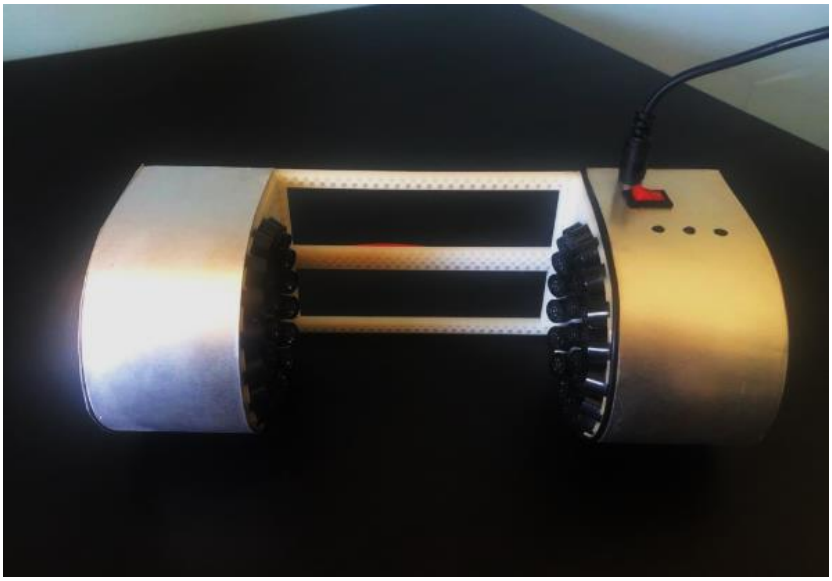
Una vez tenemos toda la electrónica dentro de la estructura de aluminio que acabamos de hacer, pasamos a colocar encima la estructura 3D con los transductores, así evitamos que se vea la electrónica y a su vez conseguimos una gran estabilidad.

En la parte superior de la estructura realizaremos la misma forma con otra lámina de aluminio de las mismas dimensiones para que quede simétrico. Esta parte superior no contará con ninguna parte electrónica en su interior, aunque servirá para cubrir los cables pelados de las patillas de los transductores.



El montaje final se ha realizado pensando en que sea posible colocarlo en muchas posiciones diferentes mostrando como el atrapamiento de partículas es estable incluso estando al revés o de lado.

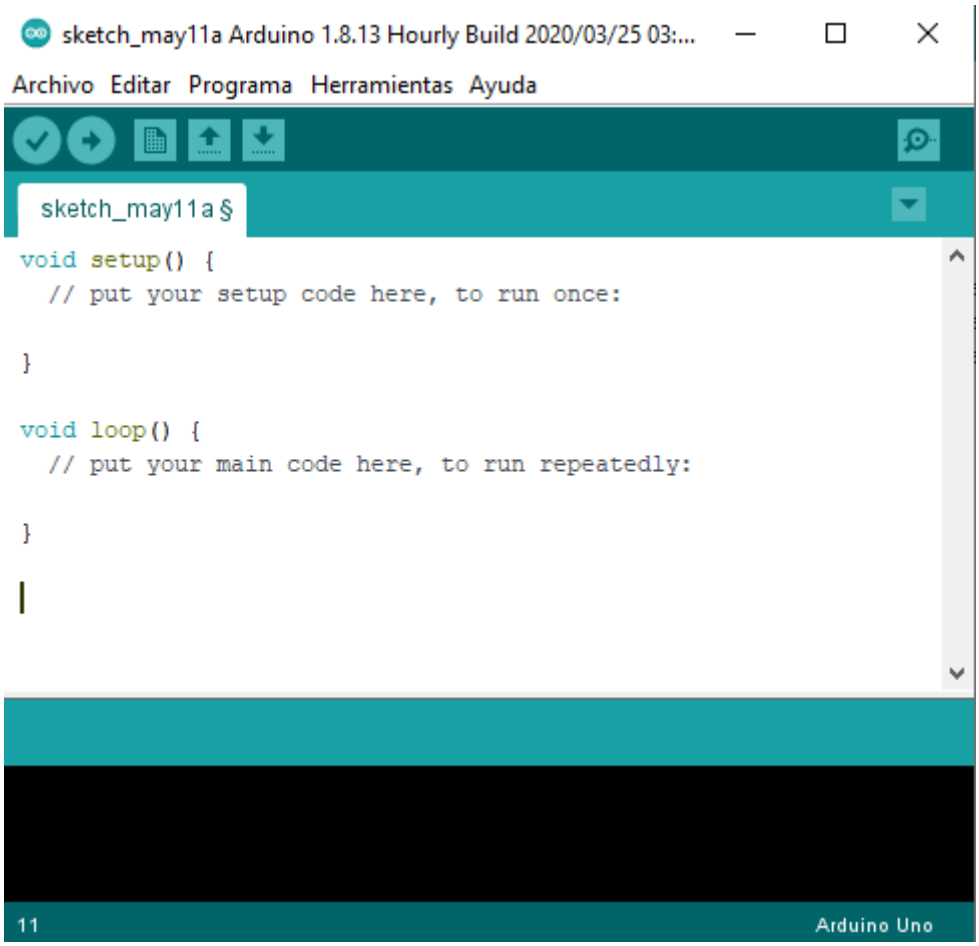




4.3.2 Programación

Una vez conectado y montado el circuito tal y como se muestra en el esquema podemos pasar a la parte de programación, que es bastante más compleja y avanzada.

Utilizaremos el software Arduino de código abierto. Se trata de un Entorno de Desarrollo Integrado (IDE) donde podremos escribir, editar y probar nuestro programa.



Esta imagen muestra la apariencia del IDE, el cual se divide (de arriba a abajo) en la barra de menús, barra de botones, editor de código, consola de mensajes y la barra de estado.

Un programa diseñado para ejecutarse sobre un Arduino (un “sketch”) siempre se compone de tres secciones:

- La sección de declaración de variables globales. Ubicada directamente al principio del sketch
- La sección void setup(). Las instrucciones aquí ubicadas se ejecutan una única vez.
- La sección void loop(). Las instrucciones aquí ubicadas se ejecutan infinitas veces hasta que la placa se apague.

Por lo tanto, generalmente en el void setup se realizan las preconfiguraciones iniciales y en el void loop las instrucciones del programa principal que se repiten continuamente.

Vamos a explicar nuestro programa siguiendo esas tres secciones principales.

Sección de declaración de variables

Comenzamos incluyendo las librerías **avr/sleep.h** y **avr/power.h**.

```
#include <avr/sleep.h>  //Administración de energía y modos de suspensión
#include <avr/power.h>  //Gestión de reducción de potencia
```

La primera es llamada "Administración de energía y modos de suspensión" y se usa para reducir considerablemente el consumo de energía, dando la opción de poner en modo suspensión la CPU mientras los relojes periféricos siguen funcionando.

La segunda se conoce como "Gestión de reducción de potencia". Muchos AVR contienen un registro de reducción de energía que le permiten reducir el consumo de energía al deshabilitar o habilitar varios periféricos integrados según sea necesario.

En la sección del void setup se encontrarán los macros para habilitar o deshabilitar los periféricos integrados para reducir la energía.

Utilizaremos la directiva especial **#define** para declarar nuestras constantes. Las constantes declaradas en Arduino no ocupan ningún espacio de memoria del programa en el chip.

```
#define N_PORTS 1
#define N_DIVS 24
```

N_PORTS se refiere al número de puertos, con los que vamos a trabajar dentro de la función macro posteriormente.

Con **N_DIVS** nos referimos a la cantidad de divisiones que encontraremos en la matriz, esto se entenderá mejor cuando expliquemos las matrices del programa.

```
#define WAIT_LOT(a) __asm__ __volatile__ ("nop"); __asm__ __volatile__ ("nop");
#define WAIT_MID(a) __asm__ __volatile__ ("nop"); __asm__ __volatile__ ("nop");
#define WAIT_LIT(a) __asm__ __volatile__ ("nop"); __asm__ __volatile__ ("nop");
```

__asm__ es una palabra clave que llama al ensamblador alineado, que permite insertar instrucciones de lenguaje ensamblado directamente en programas de lenguaje C como el de Arduino sin realizar pasos adicionales de ensamblado y sin necesidad de contar con un ensamblador independiente.

También indica que el resto de la línea es una instrucción de lenguaje ensamblado. En concreto "nop" es una instrucción de lenguaje ensamblado que no hace nada, simplemente realiza un ciclo sin operar provocando un tiempo de demora.

__volatile__es necesario para garantizar que la declaración del **__asm__** en sí no se reordena, elimina o almacena en ningún otro acceso volátil.

Es decir, cuando usamos la palabra clave **__volatile__** junto con **__asm__** lo que hacemos es decirle al compilador (optimizador) que no elimine esa parte del código, incluso aunque parezca no hacer nada. Porque para la optimización del programa se suelen eliminar las partes del código que aparentemente no tienen ninguna función, y como queremos evitar eso usamos la combinación de estas dos palabras clave.

Estamos definiendo tres constantes, las cuales contienen varias instrucciones**__asm__volatile__ ("nop");** En realidad esta instrucción no ejecutará nada porque como indica "nop", no realiza ninguna operación. Generalmente se utiliza para obtener demoras cortas, ya que en su ejecución se invierte un ciclo máquina.

Por lo tanto, cuantas más instrucciones**__asm__volatile__ ("nop");**, más tiempo de espera se conseguirá.

En la imagen solo se han incluido dos instrucciones **__asm__volatile__ ("nop");** en cada línea, pero en el programa tenemos muchas más. La cantidad exacta es la siguiente:

WAIT_LOT: Contiene 14 **__asm__volatile__ ("nop");** que producirán un tiempo de espera al que hemos llamado WAIT_LOT del inglés, Espera Mucho.

WAIT_MID: Contiene 13 **__asm__volatile__ ("nop");** por lo que hace una espera media.

WAIT_LIT: Contiene 9 **__asm__volatile__ ("nop");** por lo que hace una espera corta, como su nombre indica.

De ahí el nombre de las constantes definidas, porque en inglés tienen ese significado.

Todas las constantes las definimos con la directiva **#define**.

```
#define OUTPUT_WAVE(pointer, d)  PORTC = pointer[d*N_PORTS + 0]
```

En esta línea estamos creando una función macro, la diferencia entre un macro común y una función macro es que la función es capaz de aceptar parámetros y valores de retorno, como es el caso. Debemos tener en cuenta que todas las macros se definen en mayúsculas, por eso el nombre de nuestra función está en mayúsculas **OUTPUT_WAVE**. La ventaja que tenemos usando una macro en vez de una función real es la velocidad.

(pointer, d) son dos variables que serán los parámetros asignados en otras partes del código y la función macro será la encargada de acceder a ellos de manera correspondiente.

El asterisco está relacionado con el tema de los punteros. Los punteros son unos elementos muy potentes pero a su vez bastante complejos. En esta parte solo vamos a dar una pincelada diciendo que un puntero es un objeto del lenguaje de programación, cuyo valor se refiere a otro valor almacenado en otra parte de la memoria utilizando su dirección.

Con lo cual estamos determinando el valor que ha obtenido de acceder al array que empieza en el puntero, y lo asigna a **PORTC** que es una variable que veremos más adelante en el código. Cuando vayamos avanzando todo quedará más claro y se verán las relaciones entre todas las instrucciones explicadas.

Aquí se pueden observar más constantes las cuales se usarán más adelante en el código. Las explicaremos cuando vayamos a darles uso, para una mejor comprensión de su utilidad.

```
#define N_BUTTONS 6
#define STEP_SIZE 1
#define BUTTON_SENS 2500
#define N_FRAMES 24
```

A continuación creamos una variable estática tipo byte llamada frame asignada a 0.

```
static byte frame = 0;
```

Que sea una variable **static** significa que ha sido ubicada estáticamente y cuyo tiempo de vida se extiende durante toda la ejecución del programa. Teniendo así un ámbito más amplio que otras variables.

Es de tipo **byte**, estas variables siempre tendrán un rango de valores entre 0 y 255 y utilizarán una cantidad de memoria de un byte (8 bits) para almacenar su valor.

Lo siguiente que podemos observar en el programa es esto:

```
static byte animation[N_FRAMES][N_DIVS] =
{{0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa},
{0x9,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x6,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa},
{0x9,0x9,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x6,0x6,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa},
{0x9,0x9,0x9,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x6,0x6,0x6,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa},
{0x9,0x9,0x9,0x9,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x6,0x6,0x6,0x6,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa},
{0x9,0x9,0x9,0x9,0x9,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x6,0x6,0x6,0x6,0x6,0xa,0xa,0xa,0xa,0xa,0xa,0xa},
{0x9,0x9,0x9,0x9,0x9,0x9,0x5,0x5,0x5,0x5,0x5,0x5,0x6,0x6,0x6,0x6,0x6,0x6,0xa,0xa,0xa,0xa,0xa,0xa},
{0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x5,0x5,0x5,0x5,0x5,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0xa,0xa,0xa,0xa,0xa},
{0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x5,0x5,0x5,0x5,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0xa,0xa,0xa,0xa},
{0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x5,0x5,0x5,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0xa,0xa,0xa},
{0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x5,0x5,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0xa,0xa},
{0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x5,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0xa},
{0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6},
{0x5,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0xa,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6},
{0x5,0x5,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0xa,0xa,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6},
{0x5,0x5,0x5,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0xa,0xa,0xa,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6},
{0x5,0x5,0x5,0x5,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0xa,0xa,0xa,0xa,0x6,0x6,0x6,0x6,0x6,0x6,0x6,0x6},
{0x5,0x5,0x5,0x5,0x5,0x9,0x9,0x9,0x9,0x9,0x9,0x9,0xa,0xa,0xa,0xa,0xa,0x6,0x6,0x6,0x6,0x6,0x6,0x6},
{0x5,0x5,0x5,0x5,0x5,0x5,0x9,0x9,0x9,0x9,0x9,0x9,0xa,0xa,0xa,0xa,0xa,0xa,0x6,0x6,0x6,0x6,0x6,0x6},
{0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x9,0x9,0x9,0x9,0x9,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0x6,0x6,0x6,0x6,0x6},
{0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x9,0x9,0x9,0x9,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0x6,0x6,0x6,0x6},
{0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x9,0x9,0x9,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0x6,0x6,0x6},
{0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x9,0x9,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0x6,0x6},
{0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x9,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0x6}};
```

Vamos a ver por partes qué es y como está construido.

Estamos creando una matriz (Array variable de tipo byte), que es un conjunto de variables de un tipo concreto (byte) que tienen todas el mismo nombre pero se distinguen por un número a modo de índice. Así podemos tener una colección de valores agrupados pero pudiendo manipularlos por separado gracias a que dentro del array cada uno está identificado mediante un índice numérico.

```
static byte animation[N_FRAMES][N_DIVS]
```

Concretando un poco más se trata de un array bidimensional, es decir, es un array cuyos elementos son a su vez arrays (subarrays). Con lo cual **N_FRAMES** indica que se está declarando un array de 24 elementos donde a su vez cada uno de ellos es un subarray de otros 24 elementos (**N_DIV**).

Estos datos los sabemos porque se asignó el valor 24 a **N_FRAMES** y **N_DIV** anteriormente.

```
{0x9,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x6,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa,0xa},
```

Si contamos la cantidad de elementos en este subarray se puede ver perfectamente que son 24. Y el array principal está formado por 24 subarrays como el anterior.

Dentro de estos arrays podemos observar que los valores se dan en hexadecimal. Cuando deseamos asignar un valor en formato hexadecimal hay que preceder dicho valor mediante el prefijo “0x”.

Sección void setup()

Ya podemos pasar a esta sección, que en nuestro código es donde se realizará todo el programa. Al explicar la estructura de un sketch dijimos que esta sección se usaba simplemente para las instrucciones que se iban a ejecutar una vez, pero nosotros mediante los llamados “go to” vamos a conseguir repetir de manera infinita una parte del código que se encuentra dentro del void setup. Esto se verá claramente más adelante.

El ATmega328p como cualquier otro microcontrolador tiene registros, algunos de estos registros están relacionados con los puertos de entrada/salida, cada puerto tiene un nombre específico y sus registros asociados.

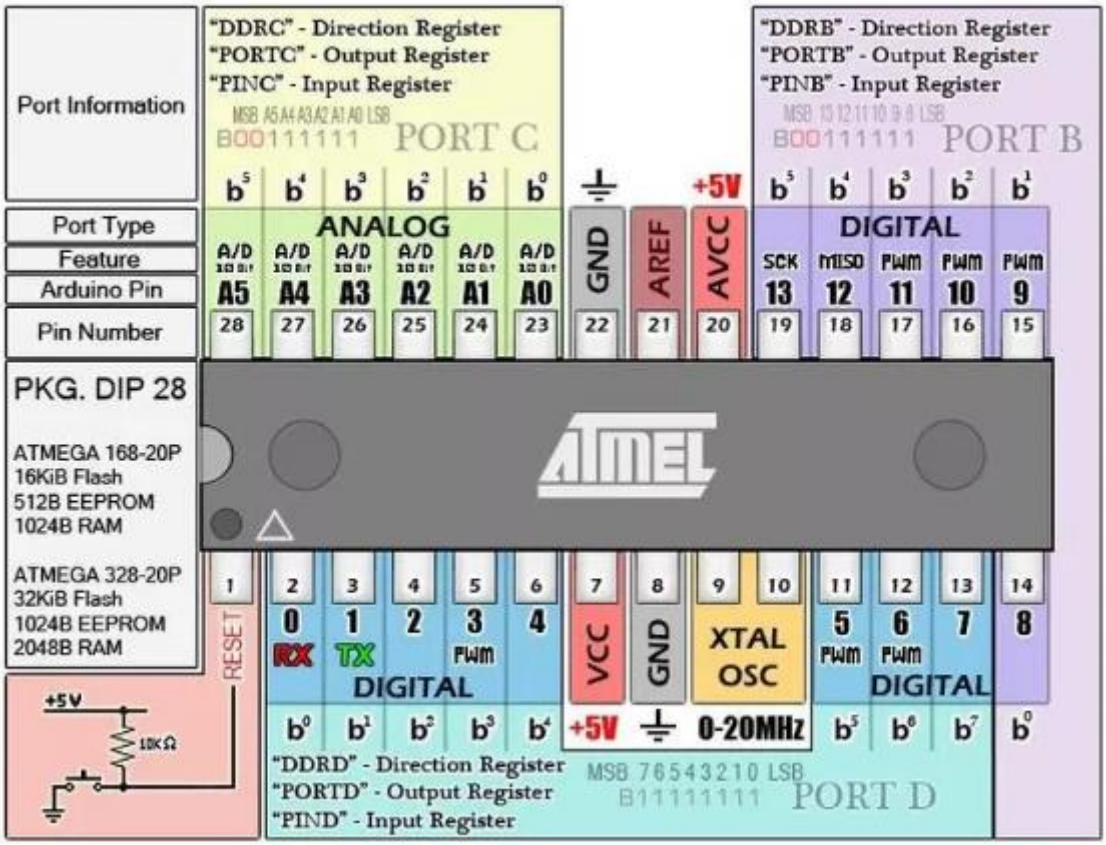
Para empezar hay un registro dedicado a cada puerto que define si un pin es de entrada o salida, que es el registro DDRX, donde X es la letra del puerto que queremos configurar. Nosotros usaremos el puerto C así que usaríamos el registro **DDRC**.

En el programa escribimos lo siguiente:

```
DDRC = 0b00001111;
```

Al utilizar registros DDR tenemos la ventaja de que con solo una instrucción podemos declarar varios pines como entrada o salida, con el clásico pinMode() necesitaríamos muchas más instrucciones y líneas de código. Cada bit en los registros DDR puede ser 1 o 0, poniéndolo a 1 configura el pin como salida y a 0 como entrada. Con lo cual, de A0 a A3 serán señales de salida.

Viendo el esquema se puede entender perfectamente, si nos fijamos en la zona del puerto C explica la información del puerto, el tipo de puerto y los pines que son. También en la información del puerto nos muestra como los dos primeros bits se deben poner a 0, y los otros seis bits corresponden a los pines A5 a A0, en ese orden de más a menos significativo.



Lo que acabamos de ver va en línea con la parte del montaje electrónico, ya que si recordamos, los pines que vamos a usar para controlar el driver serán los pines A0, A1, A2 y A3, que son los que hemos declarado como salida del Arduino.

Otro registro correspondiente al puerto C es el **PORTC** - Output Register (ver esquema anterior). Este registro **PORTC** controla si el pin está en nivel alto (1) o nivel bajo (0).

```
PORTC = 0b00000000;
```

En este caso mantenemos todos a nivel bajo.

Entendiendo esto podemos ahorrarnos muchas líneas de código, y el resultado sería el siguiente.

```
DDRC = 0b00001111;      //A0 a A3 son señales de salida
PORTC = 0b00000000;     //Pines a nivel bajo
```

Se puede ver que lo estamos configurando en binario, aunque también se podría hacer en hexadecimal cambiando el prefijo 0b por 0x.

Electrónicamente habíamos conectado con un jumper el pin 10 al pin 11, uno de ellos actuará como salida y otro como entrada de la señal de sincronismo de 40 KHz.

```
pinMode(10, OUTPUT);        //pin 10 (B2) generará una señal 40KHz a sincronismo
pinMode(11, INPUT_PULLUP); //pin 11 (B3) es el sincronismo de entrada
//IMPORTANTE tener conectado el pin 10 con el 11
```

En esta ocasión hemos usado el **pinMode()** para configurar si queremos que el pin especificado se comporte como entrada (**INPUT**) o salida (**OUTPUT**). Se ha usado porque así es posible habilitar las resistencias pullup internas de 20 KΩ con el modo **INPUT_PULLUP**.

```
for (int i = 2; i < 8; ++i){ //pin 2 a 7 (D2 a D7) son entradas para los botones
 pinMode(i, INPUT_PULLUP);
}
```

Ahora hemos usado otra forma para configurar si se trata de un pin de entrada o salida.

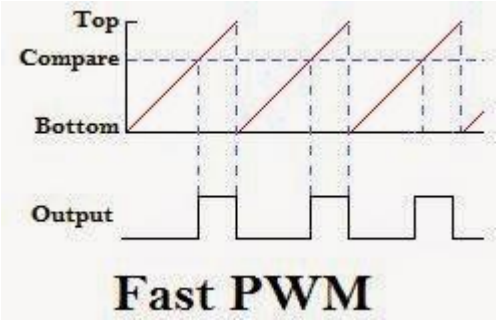
Estamos definiendo los pines D2 a D7 correspondientes a los pulsadores. Un pulsador siempre debe llevar una resistencia pullup o pulldown para evitar señales de ruido. Para ello nosotros hemos usado de nuevo las resistencias internas del Arduino con el modo **INPUT_PULLUP**.

Esto no se podría hacer usando el registro DDR, pero como queremos usar el menor número de instrucciones posibles, se ha optado por hacer un bucle for el cual se va a repetir las ocasiones necesarias como para configurar todos los pines.

Anteriormente hemos visto que configurábamos los pines analógicos A0, A1, A2 y A3 como salidas. Al tratarse de salidas analógicas entra en juego el PWM que usan los timers del microcontrolador para hacer la forma de onda PWM.

La modulación por ancho de pulsos (PWM) la usamos para modificar el ciclo de trabajo de una señal periódica sinusoidal o cuadrada, para así generar señales de audio a una frecuencia determinada.

Para generar la señal PWM utilizamos los timers configurándose en modo de trabajo **Fast PWM**, funcionando de la siguiente manera:



1. Tenemos un registro que comienza desde cero y cuenta de modo ascendente.
2. Cuando llega al nivel de comparación de otro registro el pin de salida se conmuta y se pone en nivel alto.
3. Cuando el valor del registro llega a Top vuelve a comenzar desde Bottom y el pin de salida vuelve a cambiar.
4. El tiempo que tarda el contador en llegar al final es el periodo de la señal.

Hay más modos de trabajo para generar la señal PWM, pero nosotros nos quedaremos con este.

Todo lo que necesitamos hacer en el programa es iniciar el temporizador y establecer el ciclo de trabajo. El ATmega328p tiene 3 timers (timer0, timer1 y timer2) para controlar 6 salidas PWM. Estos temporizadores generarán interrupciones cuando lleguen al nivel del registro de comparación o al Top (overflow).

En nuestro caso el control de los temporizadores se consigue con los registros de control **TCCR1A** y **TCCR1B**, los cuales tienen varios grupos de bits que debemos entender para poder realizar una correcta programación:

- **WGM** (Waveform Generation Mode bits): Bits de generación de la forma de onda. Controlan el modo general del timer.
- **CS** (Clock Select bits): Bits de selección de reloj. Permite elegir el prescaler.
- **COMxA** (Compare Match Output A Mode bits): Bits de comparación de coincidencia de salida A. Habilitan, deshabilitan e invierten la salida A.
- **COMxB** (Compare Match Output B Mode bits): Bits de comparación de coincidencia de salida B. Habilitan, deshabilitan e invierten la salida B.

Todo esto es lo necesario para comprender los registros de control. Ahora pasamos a los registros de comparación.

OCR1A y **OCR1B** (Output Compare Register) son los registros de comparación que usaremos para establecer los niveles en los que las salidas A y B se verán afectadas. Cuando el valor del temporizador coincide con el del registro, la salida correspondiente se modificará.

El código nos quedará de la siguiente manera:

```
// Genera una señal de sincronismo de 40 KHz en el pin 10
noInterrupts();           // dehabilita todas las interrupciones
TCCR1A = bit (WGM10) | bit (WGM11) | bit (COM1B1); // fast PWM, clear OC1B on compare
TCCR1B = bit (WGM12) | bit (WGM13) | bit (CS10);   // fast PWM, no prescaler
OCR1A =  (F_CPU / 40000L) - 1;
OCR1B = (F_CPU / 40000L) / 2;
interrupts();             // habilita todas las interrupciones
```

Así es como generamos una señal de 40 KHz, que es a la que trabajan los transductores.

Aunque sepamos la base del PMW, interrupciones y registros, no podemos entender esta parte del código porque todavía tenemos que saber como se configuran para conseguir la señal deseada de 40 KHz.

Lo primero es elegir con que temporizador se va a trabajar, recordemos que había tres temporizadores timer0, timer1 y timer2, nosotros lo haremos con el **timer1**. Esto se puede ver en el código de arriba (**TCCR1A/TCCR1B**). Nos hemos decantado por el timer1 que es el único temporizador de 16 bits del Arduino, lo que significa que puede almacenar un valor de contador máximo de 65535, y una vez llegue a su máximo volverá a 0.

TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Al haber seleccionado el timer1, tenemos la opción de trabajar con los pines PWM 9 (**OC1A**) y 10 (**OC1B**), en nuestro caso la señal se generará en el pin 10.

Mediante WGM como se explicó, vamos a seleccionar el modo de trabajo. Tenemos los bits **WGM10** y **WGM11** en el registro A, **WGM12** y **WGM13** en el registro B activos en nuestro programa, por lo que estamos seleccionando el modo de trabajo 15 (Fast PWM) como se puede apreciar en la tabla.

Table 13-4. Waveform Generation Mode Bit Description[(1)]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

En el programa también está claro que el bit **COM1B1** se encuentra configurado para realizar la comparación (**OC1B**) en el pin 10.

En el registro **TCCR1B** se encuentra activo únicamente el bit **CS10**, configurando así el temporizador sin prescaler:

Table 16-5. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}/1$ (No prescaling)
0	1	0	$clk_{I/O}/8$ (From prescaler)
0	1	1	$clk_{I/O}/64$ (From prescaler)
1	0	0	$clk_{I/O}/256$ (From prescaler)
1	0	1	$clk_{I/O}/1024$ (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Con todo lo que se ha visto ya se puede entender como se ha realizado la configuración de los registros de control tal y como se muestra en estas líneas de código.

```
TCCR1A = bit (WGM10) | bit (WGM11) | bit (COM1B1); // fast PWM, reinicia OC1B comparación
TCCR1B = bit (WGM12) | bit (WGM13) | bit (CS10);   // fast PWM, no prescaler
```

Para elegir el valor de comparación a partir del cual se generará el desbordamiento se calcula con la siguiente fórmula:

$$Registro\ de\ comparación = \frac{Frecuencia\ CPU}{Interrupción\ deseada\ x\ Prescaler} - 1$$

La frecuencia de reloj de Arduino es de 16 MHz.

Con esta fórmula podríamos calcular el resultado e introducirlo directamente, pero lo que hacemos es dejar al programa realizar los cálculos y ahorrarnos posibles errores de cálculos humanos.

Estas fórmulas se utilizaran para calcular el valor que recibirá cada registro de comparación:

```
OCR1A =  (F_CPU / 40000L) - 1;
OCR1B = (F_CPU / 40000L) / 2;
```

Por último, en este fragmento de código que se mostró también anteriormente aparecen varias instrucciones de interrupciones internas, que son aquellas relacionadas con los timers.

```
// Genera una señal de sincronismo de 40 KHz en el pin 10
noInterrupts();           // dehabilita todas las interrupciones
TCCR1A = bit (WGM10) | bit (WGM11) | bit (COM1B1); // fast PWM, clear OC1B on compare
TCCR1B = bit (WGM12) | bit (WGM13) | bit (CS10);   // fast PWM, no prescaler
OCR1A =  (F_CPU / 40000L) - 1;
OCR1B = (F_CPU / 40000L) / 2;
interrupts();             // habilita todas las interrupciones
```

La primera deshabilita las interrupciones, y la segunda las habilita.

Comenzamos deshabilitando las interrupciones `noInterrupts();` porque esta sección de código es crítica y nos asegura que no se realice ningún cambio mientras se ejecuta. Cuando acaba de realizar todo el proceso habilitamos nuevamente las interrupciones `interrupts();`.

El siguiente paso es deshabilitar todo lo que no vayamos a necesitar, y así conseguir la máxima eficiencia.

```
// deshabilitamos todo lo que no necesitamos
ADCSRA = 0;  // ADC
power_adc_disable ();
power_spi_disable();
power_twi_disable();
power_timer0_disable();
//power_usart0_disable();
Serial.begin(115200);
```

Comenzamos con el conversor analógico digital. Lo deshabilitamos poniendo a cero el Control de Registro A del ADC (**ADCSRA** = 0).

Ahora es cuando usamos los macros de la librería **avr/power.h** que nos permiten deshabilitar ciertos periféricos integrados que no se están utilizando.

Mediante **power_adc_disable()** deshabilitamos el módulo convertidor analógico a digital. Por lo que ya hemos deshabilitado tanto el registro como el módulo del ADC.

Usando **power_spi_disable()** deshabilitamos el módulo de interfaz periférica en serie (Serial Peripheral Interface).

Con **power_twi_disable()** deshabilitamos el módulo de interfaz de dos cables (Two Wire Interface).

El **power_timer0_disable()** deshabilita el módulo del timer0.

También en esta parte de código iniciamos la comunicación con el puerto serie a una velocidad de 115200 bits por segundo (baudios), que es la mayor que se puede usar en Arduino y de esta manera obtener la mayor cantidad de datos.

```
byte* emittingPointer = &animation[frame][0];
```

En esta línea de código se relacionan los conceptos de array y punteros, así que vamos a definir bien que es lo que está ocurriendo.

Recordemos que un puntero es una variable que sirve para almacenar la dirección de otra variable, y un array es un conjunto de variables del mismo tipo cada uno con una posición determinada. Entonces la relación que existe entre ambos conceptos es que el nombre de un array es la dirección de memoria del primer elemento del array.

Usaremos dos operadores, el asterisco (*****) para indicar a que tipo de dato apunta nuestra variable **emittingPointer**, que representa el valor contenido en la dirección deseada, y el **&** para decir a que dirección apunta.

Lo que estamos haciendo es crear un puntero llamado **emittingPointer**, y estamos almacenando en él la dirección de memoria del primer elemento del array **animation**. Este array animation ya lo vimos anteriormente.

Podemos saber que es el primer elemento porque la variable frame fue asignado al valor 0 previamente, por lo tanto, lo que está leyendo es animation[0][0] (primer valor de la matriz).

Vamos a crear una variable tipo **byte** llamada **buttonsPort** que será usada posteriormente, y le asignamos el valor 0.

```
byte buttonsPort = 0;
```

```
bool anyButtonPressed;
bool buttonPressed[N_BUTTONS];
short buttonCounter = 0;
```

Estas tres variables son de unos tipos de datos que no hemos usado hasta el momento. Las dos primeras son de tipo booleana, variables que solo pueden tener dos valores: cierto o falso. Este tipo ocupa siempre un byte de memoria.

En concreto, **buttonPressed[N_BUTTONS]** es un array de tipo **boolean** que contendrá una cantidad de variables igual al número de botones, es decir 6. Debido a que habíamos definido **N_BUTTONS** como 6.

Aquí solo estamos creando la matriz, para más adelante asignarle un valor a todas sus variables (True or False).

La variable **buttonCounter** es de tipo **short**, utiliza dos bytes de memoria para almacenarse (16 bits). Tomará el valor de 0.

Aunque seguimos en la sección de void setup vamos a comenzar con la parte del código la cual se repetirá en bucle. Esto lo conseguimos con la estructura de control **goto** que transfiere el flujo de programa a un punto del programa que está etiquetado.

Comenzamos escribiendo **LOOP**, que será la etiqueta a la que nos refiramos con el **goto**. La etiqueta debe ir seguida de dos puntos (:).

```
LOOP:
```

//La parte de código encontrada entre estos dos será la que se repita de manera indefinida.

```
goto LOOP;
```

La primera instrucción dentro del LOOP es la siguiente:

```
while(PINB & 0b00001000);
```

Utilizaremos un bucle **while** del que no saldrá hasta que la expresión del paréntesis sea falsa. Como podemos ver hemos puesto un punto y coma después de la expresión, en vez de la típica llave {, lo que estamos haciendo con esto es cerrar el while sin ninguna instrucción en su interior, con lo cual mientras sea verdadera la condición no realizará ninguna acción. Se encuentra "esperando" a que se dé otra condición para salir del bucle.

Veamos entonces cual es la condición:

Volvemos a los registros, el **PIN** es un registro al igual que **DDR** y **PORT**, pero este permite leer el estado de los pines (solo lectura). En concreto estamos usando el registro **PINB**, con el que leeremos todos los pines del puerto B al mismo tiempo (del pin 8 al 13).

El ampersand (**&**) es un operador bit a bit que hace la función de AND. Compara cada bit de su primer operando (**PINB**) con el bit correspondiente de su segundo operando (**0b00001000**). Si ambos bits son 1, el bit del resultado correspondiente se establece en 1. De lo contrario, el bit del resultado se establece en 0.

Con lo cual estamos indicando que lea todos los pines del puerto B y hasta que el pin 11 (B3) no se encuentre a nivel bajo no va a salir del bucle while. La condición será cierta mientras el pin 11 se encuentre a nivel alto.

Esta imagen está sacada del esquema mostrado en la explicación del acceso a puertos. En él se puede ver como el pin 11 del Arduino corresponde al bit B3 (b^3).

Una vez sale del while, nos encontramos con las siguientes instrucciones:

```
OUTPUT_WAVE(emittingPointer, 0); buttonsPort = PIND; WAIT_LIT();
OUTPUT_WAVE(emittingPointer, 1); anyButtonPressed = (buttonsPort & 0b11111100) != 0b11111100; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 2); buttonPressed[0] = buttonsPort & 0b00000100; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 3); buttonPressed[1] = buttonsPort & 0b00001000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 4); buttonPressed[2] = buttonsPort & 0b00010000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 5); buttonPressed[3] = buttonsPort & 0b00100000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 6); buttonPressed[4] = buttonsPort & 0b01000000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 7); buttonPressed[5] = buttonsPort & 0b10000000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 8); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 9); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 10); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 11); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 12); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 13); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 14); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 15); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 16); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 17); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 18); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 19); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 20); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 21); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 22); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 23);
```

Vamos a analizar los elementos que conforman esta parte del código y la relación entre ellos:

OUPUT_WAVE fue declarada en la sección de declaración de variables.

Nos encontramos con 24 OUTPUT_WAVE para recorrer todos los valores del array animation, haciendo un barrido completo de la matriz.

El primer parámetro, **emittingPointer**, recordemos que es un puntero que apunta al array animation, el segundo parámetro son números que van desde el 0 al 23, es decir, un total de 24 que es igual a la cantidad de subarrays de la matriz animation.

```
OUTPUT_WAVE(emittingPointer, 0);
```

Anteriormente definimos lo botones del pin D2 a D7 como entradas. Como estos pertenecen al Puerto D del ATmega328p lo que hacemos es leer el valor de los pines mediante el registro **PIND**, cuya lectura guardaremos en la variable **buttonsPort**.

En este momento se produce una espera corta debido a **WAIT_LIT**.

Entonces la primera línea de esta parte del código queda de la siguiente manera:

```
OUTPUT_WAVE(emittingPointer, 0); buttonsPort = PIND; WAIT_LIT();
```

Vamos a poner un ejemplo sobre como se podría haber hecho en varias líneas, pero el código quedaría muy largo, cosa que queremos evitar.

```
OUTPUT_WAVE(emittingPointer, 0);
buttonsPort = PIND;
WAIT_LIT();
```

Esto es justamente lo mismo que hemos visto en una sola línea. Puede que quede algo más visual, aunque nosotros estamos buscando la eficiencia y ahorrar líneas de código innecesarias, así optamos por la primera opción.

La siguiente línea de esta parte comienza igual que la anterior, aunque con un “1” como segundo argumento de OUTPUT_WAVE.

Aquí usamos por primera vez la variable booleana **anyButtonPressed** que habíamos declarado sin asignarle ningún valor. Mediante el operador bit a bit **&** (AND) hacemos la operación entre **buttonsPort** y **0b11111100**, este operador fue explicado con mayor detalle anteriormente.

0b11111100 corresponde a los pines del D2 al D7, se observa que los dos bits menos significativos se encuentran a 0, esto es debido a que son los pines encargados de la transmisión y recepción de datos del Arduino (D1/**TX** y D0/**RX**). Si no se pusiesen a 0, el pin D0 podría producir errores no deseados y el puerto serie dejaría de funcionar.

!= es otro operador que compara la variable de la izquierda con el valor de la derecha del operador. Estamos comparando el resultado de la operación **buttonsPort & 0b11111100** con el valor **0b11111100**. Devuelve verdadero cuando los dos operandos no son iguales, estableciéndose **anyButtonPressed** como True.

En este momento se produce una espera media debido a **WAIT_MID**.

```
OUTPUT_WAVE(emittingPointer, 1); anyButtonPressed = (buttonsPort & 0b11111100) != 0b11111100; WAIT_MID();
```

Seguidamente nos encontramos seis líneas de código, que van desde el 2 al 7 del segundo argumento, que equivalen a las posiciones 3 al 8 de la matriz. Donde se generarán unas ondas determinadas OUTPUT_WAVE.

Aquí entra en juego la matriz **buttonPressed** de tipo booleana. Este array tiene 6 posiciones donde se asignarán ciertos valores (0 o 1).

La primera línea corresponde a la primera posición del array, **buttonPressed[0]**, a la que se le asigna un valor resultado de una operación. La operación (**&**) es una función AND que tiene como operandos **buttonsPort y 0b00000100**. Recordemos que **buttonsPort** contiene la lectura de los pines del puerto D, que leímos mediante **PIND**. Y el otro operando en binario se está refiriendo al pin D2.

Comparará los bits de los dos opernados y si ambos bits son 1 el resultado será 1, quedando guardado ese valor en la primera posición del array **buttonPressed**.

En este momento se produce una espera media debido a **WAIT_MID**.

Este procedimiento se realiza seis veces, pasando por todos los pines (D2 a D7) y guardando el valor en la posición correspondiente de la matriz **buttonPressed**.

```
OUTPUT_WAVE(emittingPointer, 2); buttonPressed[0] = buttonsPort & 0b00000100; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 3); buttonPressed[1] = buttonsPort & 0b00001000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 4); buttonPressed[2] = buttonsPort & 0b00010000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 5); buttonPressed[3] = buttonsPort & 0b00100000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 6); buttonPressed[4] = buttonsPort & 0b01000000; WAIT_MID();
OUTPUT_WAVE(emittingPointer, 7); buttonPressed[5] = buttonsPort & 0b10000000; WAIT_MID();
```

Las siguientes 15 líneas actúan todas de la misma forma.

Primero con **OUTPUT_WAVE**, al igual que en los casos anteriores, seleccionamos la onda de salida recorriendo los valores del array animation gracias a los dos argumentos (**emittingPointer, x**), donde x son los números de 8 a 22.

Después se realiza una espera más larga usando **WAIT_LOT**.

Aquí no estamos leyendo botones ni usando operadores. Simplemente nos dedicamos a generar la onda de salida.

```
OUTPUT_WAVE(emittingPointer, 8); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 9); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 10); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 11); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 12); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 13); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 14); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 15); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 16); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 17); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 18); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 19); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 20); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 21); WAIT_LOT();
OUTPUT_WAVE(emittingPointer, 22); WAIT_LOT();
```

La última línea de esta parte que estamos analizando es simplemente un **OUTPUT_WAVE** con sus correspondientes argumentos.

Con esta ya hemos barrido los 24 subarrays de animation.

```
OUTPUT_WAVE(emittingPointer, 23);
```

En resumen, lo que estamos haciendo es generar una onda, que varía dependiendo del subarray con el que se está trabajando. Se realizan ciertas operaciones y comparaciones para ver el estado de los botones y seguidamente se realiza una espera de un determinado tiempo.

Comenzamos con la parte de las sentencias condicionales, donde contaremos con estructuras condicionales simples y anidadas.

```
if( anyButtonPressed ){      //Si algún botón pulsado
   ++buttonCounter;          //Contador ++
   if (buttonCounter > BUTTON_SENS){    //Si contador es mayor a BUTTON_SENS (2500)
    buttonCounter = 0;                  //Contador a 0

    if (! buttonPressed[0] ) {    //Si buttonPressed[0] no es True
      if( frame < STEP_SIZE ) {   //Si frame es menor a STEP_SIZE (1)
        frame = N_FRAMES-1;       //frame será igual a N_FRAMES (24) - 1
     }else{                       //Si no
        frame-=STEP_SIZE;         //frame será frame + STEP_SIZE (1)
     }
    }
    else if (! buttonPressed[1] ) {          //Si buttonPressed[1] no es True
      if ( frame >= N_FRAMES-STEP_SIZE ) {   //Si frame es mayor o igual a N_FRAMES-STEP_SIZE (24 - 1)
        frame = 0;                           //Ponemos frame a 0
      }else {                                //Si no
        frame+=STEP_SIZE;                    //frame será frame + STEP_SIZE (1)
      }
   }
   else if (! buttonPressed[2] ) {           //Si buttonPressed[2] no es True
      frame = 0;                              //Ponemos frame a 0
   }

    emittingPointer = & animation[frame][0];  //Selecciona la posición de la matriz
   }
}else {                  //Si no
  buttonCounter = 0;     //Contador a 0
}
```

Lo primero es un **if** donde la condición es **anyButtonPressed**, si es True se ejecuta el conjunto de declaraciones de su interior, si no pasaría a ejecutar el bloque del **else**.

El valor de la variable booleana **anyButtonPressed** se consiguió de la instrucción que explicamos en la parte de los **OUTPUT_WAVE**: `anyButtonPressed = (buttonsPort & 0b11111100) != 0b11111100;`

Si se da la condición del **if** principal pasaría a sumar 1 al contador, mediante la instrucción **++buttonCounter**. El "++" es un operador que incrementa el valor de la variable en 1.

```
if( anyButtonPressed ){
   ++buttonCounter;
```

Dentro de este **if** principal nos encontramos otro **if** que tiene como condición **buttonCounter > BUTTON_SENS**, por lo que no entrará en ese bucle hasta que el contador sea mayor a 2500, que es el valor que le dimos a **BUTTON_SENS** en la sección de declaración de variables.

Una vez dentro lo primero que hace es poner el contador a 0.

```
if (buttonCounter > BUTTON_SENS){
 buttonCounter = 0;
```

Hasta el momento sabemos que lo que haremos es ver si hay algún botón pulsado y si lo hay comenzará a incrementar en 1 el contador hasta que llegue a los 2500, que será cuando entre en el **if** donde se vuelve a poner a 0 el contador, y seguiremos realizando las instrucciones que se encuentran dentro de este.

Dentro de este segundo **if** nos encontramos con otro **if** y dos **else if**, con sus respectivos **else** en las dos primeras sentencias condicionales. Estos corresponderán a los tres botones que utilizaremos para controlar el movimiento de las partículas. Durante el programa siempre hemos hablado de seis botones, aunque por el momento con tres tenemos suficientes. Si en el futuro se desea añadir una función adicional se podría perfectamente ya que tenemos todo preparado para usar los seis.

El primer **if** corresponde al botón 0. La condición dice que si **buttonPressed[0]** es 0 (False) realizará las instrucciones de su interior. Esto se debe al operador booleano ! (NO).

```
if (! buttonPressed[0] ) {
```

Antes de ver lo que encontramos en el interior de esta sentencia, recalcar que todos los valores de **buttonPressed** los asignamos en la parte de OUTPUT_WAVE:

```
buttonPressed[0] = buttonsPort & 0b00000100;
buttonPressed[1] = buttonsPort & 0b00001000;
buttonPressed[2] = buttonsPort & 0b00010000;
buttonPressed[3] = buttonsPort & 0b00100000;
buttonPressed[4] = buttonsPort & 0b01000000;
buttonPressed[5] = buttonsPort & 0b10000000;
```

Si entramos en la sentencia del botón 0 hay otro **if** con la condición **frame < STEP_SIZE**, es decir, que solo realizará lo que hay en su interior si la variable frame es menor que 1 (valor de STEP_SIZE).

La instrucción del interior de este **if** lo que hace es igualar **frame** a **N_FRAMES** menos 1, donde **N_FRAMES** es 24, con lo cual frame será igual a 23.

Si **frame** no es menor a **STEP_SIZE** (1), realizará el **else**, que va restando **STEP_SIZE** al valor de **frame**.

```
if (! buttonPressed[0] ) {
  if( frame < STEP_SIZE ) {
    frame = N_FRAMES-1;
 }else{
    frame-=STEP_SIZE;
 }
}
```

Lo que estamos consiguiendo es una cuenta descendente de 23 a 0, cuando llega a 0 el **if** será cierto y volverá a poner **frame** a 23. Así bajamos la partícula que se encuentra levitando.

Vemos que en el **else** tenemos la siguiente instrucción:

```
frame-=STEP_SIZE;
```

El operador −= es un operador compuesto y lo que hace es abreviar una resta. Su equivalente sin abreviar sería **frame = frame – STEP_SIZE**.

Ahora pasamos al **else if** donde tenemos la misma condición que en el **if** anterior solo que con el botón 1 (**!buttonPressed[1]**), es decir, si el valor de este es 0 realiza lo que se encuentra en su interior, que es otro **if** con su correspondiente **else**.

El **if** será cierto si **frame** es mayor o igual a **N_FRAMES** menos **STEP_SIZE** (24 – 1 = 23). Cuando nos encontramos con que **frame** tiene un valor superior a 23 realiza la instrucción del **if**, **frame = 0** que le da un valor de 0 a la variable.

Mientras se mantenga inferior a 23 sumará el valor de **STEP_SIZE**(1) a **frame**. Vemos que la instrucción a la que nos referimos dentro del **else** es **frame+=STEP_SIZE**, la cual cuenta con un operador compuesto de suma += que equivale a **frame = frame + STEP_SIZE**.

```
else if (! buttonPressed[1] ) {
  if ( frame >= N_FRAMES-STEP_SIZE ) {
    frame = 0;
  }else {
    frame+=STEP_SIZE;
  }
```

Por lo cual con este **else if** perteneciente al botón 1 se está realizando una cuenta ascendente de 0 a 23, cuando llega a 23 el **if** será cierto y volverá a ponerse a 0. Es justo al contrario que en el caso anterior.

El segundo y último **else if**, correspondiente al botón 2, encontramos como condición **!buttonPressed[2]**, lo que hace que entre la sentencia si su valor es 0, ya que está el operador !.

Dentro de este **else if** solo hay una instrucción y es la de **frame = 0**, lo que asignará el valor de 0 a la variable frame.

```
else if (! buttonPressed[2] ) {
   frame = 0;
}
```

Como vemos no cuenta con ningún **else**, en este caso no lo necesitaremos.

Esté botón será usado como reinicio y volverá a la configuración de onda inicial, así si hemos tocado los otros botones y queremos volver al estado de inicio solo tendríamos que pulsar este botón y así no necesitamos apagar y encender el aparato cada vez.

Una vez hemos visto estas tres sentencias condicionales de los 3 botones solo queda una instrucción encontrada dentro del **if** secundario que estamos analizando, `if (buttonCounter > BUTTON_SENS){`.

Esta instrucción es la de `emittingPointer = & animation[frame][0];`. Vemos que estamos usando el puntero **emittingPointer** al que le asignamos un valor concreto en la variable **frame**. Como ya está explicada la función de este puntero, basta con decir que la posición que tome en la matriz **animation** dependerá del valor **frame**, el cual se va modificando dependiendo del botón que estemos pulsando.

Lo que haremos será recorrer la matriz de manera ascendente, descendente o poniéndola a 0 con el uso de los pulsadores y así poder controlar en que posición queremos la partícula, controlándola mediante ese ascenso o descenso de posiciones de la matriz.

Así en el caso de encontrarnos en la parte ascendente comenzará a subir por la matriz de 1 en 1, cambiando de ondas emitidas y desplazando así la partícula levitada hacia arriba sin dar saltos bruscos que produciría que la partícula saliese de uno de sus nodos y cayese de forma inevitable.

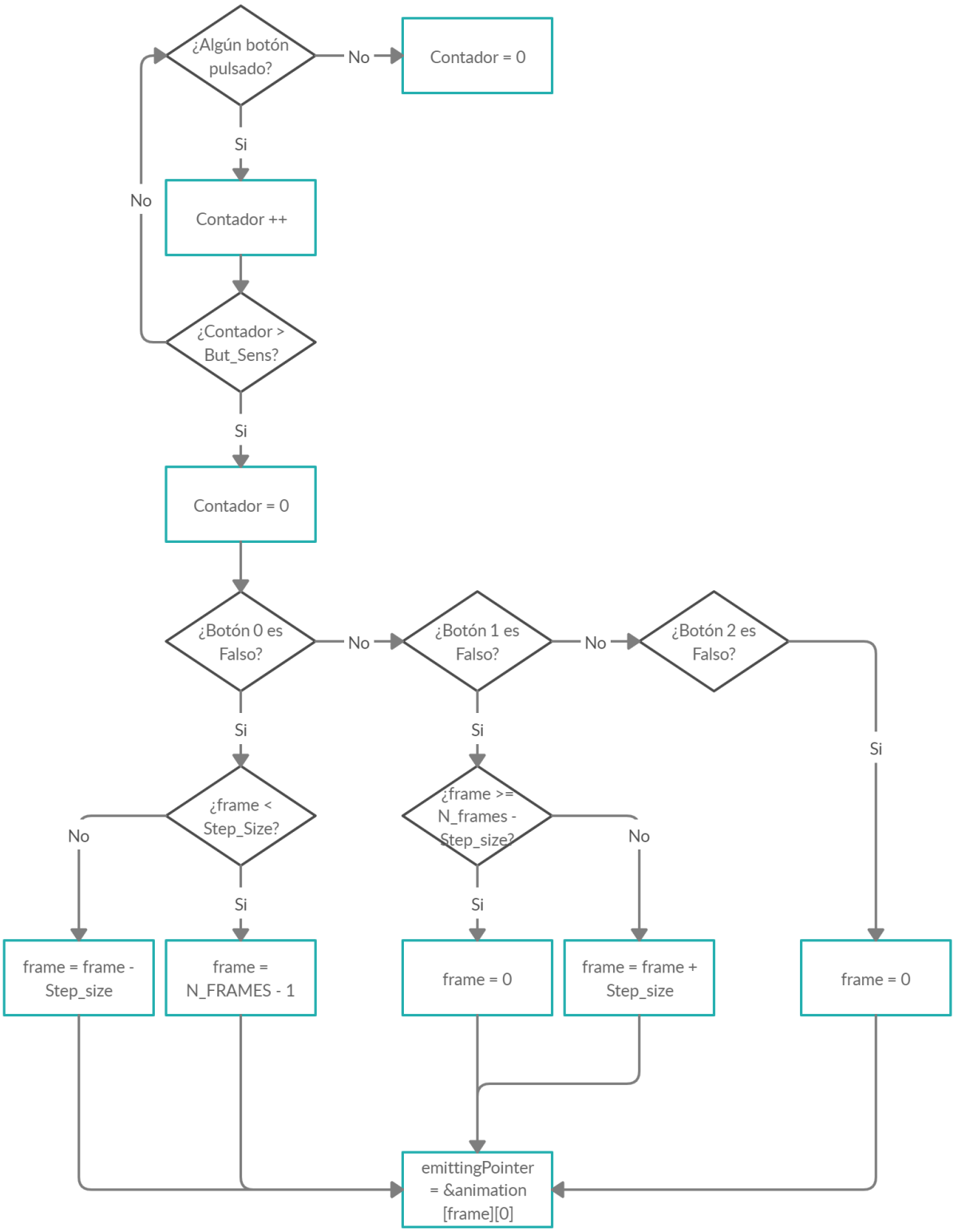
En el caso descendente sería de la misma manera.

Y en el caso del botón 2 realizamos un reinicio, el cual hay que controlar bien para que el cambio de ondas no sea demasiado grande.

Finalmente nos encontramos con el **else** que realizaría si el **if** principal no es cierto. El **else** solo tiene una instrucción y es la de poner el contador (**buttonCounter**) a 0.

```
else {
  buttonCounter = 0;
}
```

Con la ayuda del diagrama de flujo de la siguiente página es mucho más visual y fácil de comprender.



Seguidamente se encuentra el **goto LOOP**; que se explicó con anterioridad.

<u>Sección void loop()</u>

Esta sección no la utilizamos debido al uso del goto.

5. Conclusiones

Se realizó un levitador no resonante de eje único con un coste de fabricación económico.

Se explicaron todos los conceptos físicos que intervienen en el proceso de levitación, que tipo de levitaciones existen y porqué la acústica es la mejor para nuestros objetivos.

El empleo del método matricial con geometría cóncava de 4,5 cm de radio resultó ser la más efectiva según los datos recogidos en los estudios y simulaciones hechos por ordenador.

Al realizar el análisis cuantitativo se calculó que el tamaño de la partícula a levitar sería de 4 mm debido a que es del tamaño medio de la longitud de onda producida por los transductores de 40 KHz. Estos se colocaron en círculos de 6, 12 y 18 emisores, formando una matriz superior e inferior separados a una distancia de 6 cm. Con estas matrices de transductores acústicos evitamos utilizar bocinas de Langevin tradicionales.

La realización de un levitador no resonante con transductores permite trabajar en un rango de temperatura bastante amplio sin perder fuerza de atrapamiento de partículas, disminuye los niveles de voltaje requeridos y simplifica el proceso de montaje comparado con un proyecto de bocinas Langevin.

Mediante pruebas del levitador se demostró que funciona adecuadamente durante períodos prolongados de tiempo.

La programación es una parte fundamental del proyecto, por lo que se explicó con mucho detalle, así como la manera en la que se generan las ondas para poder conseguir los nodos en los que la energía será suficiente para levitar las partículas.

Se consiguió acceder directamente a los puertos de nuestro microcontrolador para tener una reacción más rápida utilizando menos ciclos de reloj y necesitando muchos menos bytes de código para hacer el mismo trabajo.

Diseñamos y montamos una estructura estable con una estética profesional que le dio al proyecto la posibilidad de adoptar diferentes posiciones con resultados óptimos.

En cuanto a aportaciones del proyecto cabe destacar la forma diferente de abordar la problemática de accesibilidad de proyectos de levitación acústica para el usuario medio, y cabe destacar el enfoque exhaustivo en entender todo lo que engloba este tipo de proyectos.

Este trabajo es una democratización de la levitación acústica previamente limitada a unos pocos laboratorios de investigación.

6. Líneas de investigación futuras

En cuanto a posibles líneas de investigación futuras se podrían cambiar algunos aspectos del modelo actual para que sea más potente y pueda levitar partículas de mayor tamaño. Para obtener mayor potencia utilizaríamos transductores de 16 mm en lugar de los de 10 mm usados en este proyecto, y agregaríamos otro anillo exterior de 24 transductores. Si quisiéramos levitar partículas de mayor tamaño necesitaríamos transductores de 25 KHz, los cuales generan una longitud de onda mayor debido a que la frecuencia es inversamente proporcional a la longitud de onda. Este levitador podría llegar hasta los 20 V en el driver y así levitar partículas de mayor densidad. También existe la posibilidad de agregar nuevas funciones con la ayuda de pulsadores adicionales.

El proceso de fabricación sería el mismo que se ha llevado a cabo para este proyecto aunque variando el tamaño de la estructura impresa 3D para adaptarse a transductores de 16 mm. Dependiendo del tipo de

transductores necesitaremos una distancia entre matrices mayor o menor. Todos los modelos de las estructuras pueden ser encontrados en proyectos de código libre.

Como se aprecia en el título del proyecto, nosotros realizamos un control de eje único. Una modificación más compleja sobre este modelo sería agregar otras dos matrices de transductores en los laterales para conseguir el control de la partícula en los tres ejes (x, y, z), las cuatro matrices que conformarían el levitador tendrían que ser de forma plana. Para conseguir estas modificaciones nos podemos basar en modelos aportados por los científicos Yoichi Ochiai, Takayuki Hoshi y Jun Rekimoto de la universidad de Tokio.

El hecho de que la levitación acústica sea un campo de estudio relativamente reciente permite que exista una gran variedad de posibles futuros diseños que incorporen mejoras con respecto a las prestaciones actuales de este proyecto.

Una evolución de este proyecto es el rayo tractor, el cual es un dispositivo formado por una sola matriz de transductores emitiendo un haz acústico que genera un campo de fuerza capaz de atrapar objetos atrayéndolos hacia la fuente de emisión.

Este avance tiene un gran potencial en la medicina, pues el sonido viaja mucho mejor en líquidos que en sólidos, lo cual permitirá conseguir muy buenos resultados moviendo partículas en el interior del cuerpo. Por ejemplo se está estudiando la posibilidad de mover piedras en el riñón a un lugar en el que no causasen problemas ni molestias.

En la química farmacéutica se podrán realizar muchos avances gracias a este tipo de proyectos. Para saber como es capaz de ayudar se debe entender que las estructuras de los fármacos a nivel atómico pueden ser amorfas o cristalinas. Los primeros son los que tienen mayor facilidad de absorción por el organismo, por lo que necesitaríamos una menor dosis de un medicamento que con la estructura cristalina. Sin embargo, casi todos los fármacos comerciales tienen la peor opción, la cristalina. Esto es debido a que las estructuras amorfas se evaporan rápidamente al contacto con el recipiente y tiende a cristalizarse.

Lo que permitiría la leviación acústica es estudiar el fármaco mientras se encuentra con estructura amorfa. Esto es lo que están haciendo por ejemplo en el Argonne National Laboratory, donde ya han investigado con esta técnica más de una docena de productos farmacéuticos.

En el ámbito industrial también tiene un gran abanico de posibilidades. Recientemente, el 4 de febrero de 2020 se publica un artículo respaldado por la Fundación de Investigación de São Paulo sobre el primer pick-and-place sin contacto gracias a la levitación acústica. Es capaz de recoger, autocentrar, orientar, transportar y liberar un dispositivo sin ningún tipo de contacto, evitando así la contaminación y el rayado de componentes frágiles. En otros métodos, cuando las pinzas o ventosas recogen los objetos, siempre existe el riesgo de contaminación o de modificar su superficie. Esto puede llegar a ser una alternativa interesante para manipular objetos delicados como dispositivos microelectromecánicos, matrices de silicio o dispositivos microópticos. (A. B. Andrade, S. Ramos, C. Adamowski, & Marzo Pérez, 2020)

7. Temporalización

- Montaje: 30 horas
- Programación: 15 horas
- Investigación: 150 horas
- Memoria: 80 horas
- PowerPoint: 10 horas

8. Bibliografía

A. B. Andrade, M., S. Ramos, T., C. Adamowski, J., & Marzo Pérez, A. (04 de 02 de 2020). Contactless pick-and-place of millimetric objects using inverted near-field acoustic levitation. *Applied Physics Letters*. doi:10.1063/1.5138598

Gregory, M., & Glynn, H. (1 de Abril de 2000). Measurement of Aqueous Foam Rheology by Acoustic Levitation. *Physical Review E, 61*. doi:10.1103/PhysRevE.61.R2204

Jackson, H., Barmatz, M., & Wagner, P. (1994). Microwave Tower Absorption Profile in a Cylindrical Sample Contained in a Resonant Cylindrical Cavity. *MRS Proceedings*, 317-347. doi:10.1557/PROC-347-317

Jared, S. (12 de Septiembre de 2012). No magic show: Real-world levitation to inspire better pharmaceuticals. *Argonne National Laboratory*.

Marzo Pérez, A. (2015). Holographic Acoustic Elements for Manipulating Levitating Particles. *Nature Communications*. doi:10.1038/ncomms9661

Marzo Pérez, A., Barnes, A., & Drinkwater, B. W. (2017). TinyLev: A multi-emitter single-axis acoustic levitator. *Review of Scientific Instruments, 88*. doi:10.1063/1.4989995

Ochiai, Y., Hoshi, T., & Rekimoto, J. (9 de Julio de 2014). Three-Dimensional Mid-Air Acoustic Manipulation by Ultrasonic Phased Arrays. *The PLOS ONE Staff , 9*(7). doi:10.1371/journal.pone.0102525

Octavio Medrano, D., & Felipe Ruiz, D. (9 de Diciembre de 2010). Diseño y construcción de un levitador acústico, mediante el principio de onda estacionaria, aplicando transporte sin contacto. *Universidad De San Buenaventura*

Oran, W. A., Berge, L. H., & Parker, H. W. (1980). Parametric study of an acoustic levitation system. *Review of Scientific Instruments, 51*(5), 626-631.

Torrente Artero, Ó. (2016). *El mundo GENUINO-ARDUINO.* RC Libros.

Valencia Murillo, R. A. (Julio de 2017). Sistema de levitación ultrasónica para análisis espectroscópico de muestras líquidas. México.

Whymark, R. R. (1 de Octubre de 1975). Levitation of Objects Using Acoustic Energy. *NASA Tech Brief*.

Xie, W. J., & Wei, B. (2001). Parametric study of single-axis acoustic levitation. *Applied Physics Letters, 79*(6), 881-883.

Xie, W. J., & Wei, B. (2002). Dependence of acoustic levitation capabilities on geometric parameters. *Physical Review E, 66*(2), 026605.

Xie, W. J., & Wei, B. (2003). Temperature dependence of single-axis acoustic levitation. *Journal of Applied Physics, 93*(5), 3016-3021.

Xie, W. J., Wei, B., Cao, C. D., Lü, Y. J., & Hong, Z. Y. (2006). Acoustic method for levitation of small living animals. *Applied Physics Letters, 89*. doi:10.1063/1.2396893

9. Webgrafía

Arduino. (19 de Febrero de 2019). Obtenido de https://www.arduino.cc/reference/en/language/structure/compound-operators/compoundsubtraction/

Arduino. (19 de Febrero de 2019). Obtenido de https://www.arduino.cc/reference/en/language/structure/control-structure/goto/

Arduino. (Sin fecha). Obtenido de https://www.arduino.cc/en/Reference/PortManipulation

Centro Superior de Innovación Educativa. (13 de Septiembre de 2016). *upnatv*. Obtenido de https://upnatv.unavarra.es/pub/rayo-tractor-asier-marzo

Crasseux. (Sin fecha). Obtenido de http://crasseux.com/books/ctutorial/Macro-functions.html

Cruz, A. (17 de Mayo de 2014). *Electronilab*. Obtenido de https://electronilab.co/tutoriales/tutorial-de-uso-driver-dual-l298n-para-motores-dc-y-paso-a-paso-con-arduino/

David. (25 de Enero de 2013). *Nongnu*. Obtenido de https://www.nongnu.org/gccintro-es/gccintro.es.html

Forum, A. (Septiembre de 2017). Obtenido de https://forum.arduino.cc/index.php?topic=497994.0

Forum, A. (Sin fecha). Obtenido de https://www.arduino.cc/en/pmwiki.php?n=Reference/Static

Forum, A. (Sin fecha). Obtenido de https://www.arduino.cc/en/Tutorial/InputPullupSerial

Gammon, N. (8 de Enero de 2012). Obtenido de http://gammon.com.au/interrupts

Ghassaei, A. (Sin fecha). *Instructables*. Obtenido de https://www.instructables.com/id/Arduino-Timer-Interrupts/

Girl, P. (13 de Diciembre de 2017). Obtenido de https://www.youtube.com/watch?v=ABjRnSYw-4k&feature=emb_logo

Jecrespom. (8 de Noviembre de 2016). *Aprendiendo Arduino*. Obtenido de https://aprendiendoarduino.wordpress.com/2016/11/08/entradas-y-salidas-arduino/

Jecrespom. (13 de Noviembre de 2016). *Aprendiendo Arduino*. Obtenido de https://aprendiendoarduino.wordpress.com/2016/11/13/interrupciones/

Laboratory, A. N. (12 de Septiembre de 2012). Obtenido de https://www.youtube.com/watch?v=669AcEBpdsY

Madrid, A. C. (1 de Marzo de 2013). Obtenido de https://www.youtube.com/watch?v=5grojEbMXjc

Madrid, A. C. (2 de Marzo de 2013). Obtenido de https://www.youtube.com/watch?v=-BV-iUvm_rg

Marzo Pérez, A. (10 de Agosto de 2017). Obtenido de ftp://ftp.aip.org/epaps/rev_sci_instrum/E-RSINAK-88-042708

Microsoft. (4 de Noviembre de 2016). Obtenido de https://docs.microsoft.com/es-es/cpp/c-language/inline-assembler-c?view=vs-2019

Microsoft. (29 de Enero de 2018). Obtenido de https://docs.microsoft.com/es-es/cpp/c-language/c-bitwise-operators?view=vs-2019

Microsoft. (9 de Octubre de 2018). Obtenido de https://docs.microsoft.com/es-es/cpp/assembler/inline/asm?view=vs-2019

Multicomp. (15 de Octubre de 2013). *Farnell*. Obtenido de http://www.farnell.com/datasheets/1759976.pdf?_ga=2.186794735.698598588.1587466494-792188664.1587466494

Navarratv. (28 de Febrero de 2019). Obtenido de https://www.youtube.com/watch?v=Up702h2saIM

Nongnu. (8 de Febrero de 2016). Obtenido de https://www.nongnu.org/avr-libc/user-manual/group__avr__sleep.html

Nongnu. (8 de Febrero de 2016). Obtenido de https://www.nongnu.org/avr-libc/user-manual/group__avr__power.html

Rafael. (26 de 06 de 2017). *Loud Cave*. Obtenido de https://loudcave.es/hablemos-sobre-el-sonido-levitacion-acustica/

Stackoverflow. (Octubre de 2014). Obtenido de https://stackoverflow.com/questions/26456510/what-does-asm-volatile-do-in-c

Upna. (5 de Octubre de 2016). Obtenido de https://www.youtube.com/watch?v=Qlq5clzE9qQ&feature=youtu.be

UpnaLab. (17 de Agosto de 2017). Obtenido de https://www.youtube.com/watch?time_continue=64&v=yVDWrWpaBho&feature=emb_logo

UpnaLab. (2017). *Instructables*. Obtenido de https://www.instructables.com/id/Acoustic-Levitator/

Wek. (18 de Abril de 2011). *AVR Freaks*. Obtenido de https://www.avrfreaks.net/forum/few-remarks-avr-gcc-inline-assembler?skey=inline%20assembler%20tutorial

Wiener-Bronner, D. (3 de Enero de 2014). *The Atlantic*. Obtenido de https://www.theatlantic.com/technology/archive/2014/01/japanese-scientists-take-levitation-next-level/356684/