



UNIVERSIDAD NACIONAL DE CÓRDOBA
FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

ELECTRÓNICA DIGITAL III

TRABAJO FINAL:

“ Snake Game”

GRUPO 9

Link repositorio:

<https://github.com/EzeErlicher/EDIIIrepo>

Integrantes:

- BADARIOTTI, Juan Manuel - 42260003
- ERLICHER, Ezequiel - 42051917

Docentes:

- SÁNCHEZ, Julio
- GALLARDO, Fernando

Descripción general del trabajo	3
Lógica y mecánicas del juego	4
main():	5
initGame():	6
moveSnake()	7
checkCollisions()	8
updateDirection()	9
createNewApple()	9
getRandomPair()	10
helloWorld() y sendStats()	10
render()	12
stopGame()	13
Matriz Led 8x8 y pines que la controlan	14
Botones de movimiento y Start/Restart(GPIO)	16
Regulador de velocidad (ADC y potenciómetro)	17
Reproducción de audio mediante DMA y DAC	19
Cálculos:	19
Configuración:	20
Envío de estadísticas y control de movimiento por UART 1	21
Diagrama de conexiones en datasheet	24
Anexos	25

Descripción general del trabajo

El trabajo final realizado, consistió en la construcción del tradicional Snake game haciendo uso de la placa LPC1769 (versión revB) con procesador ARM Cortex-M3. Dicho juego, el cual muestra sobre una matriz LED 8x8, posee las siguientes funcionalidades:

- Regulador de velocidad de movimiento mediante un potenciómetro y módulo ADC
- Control de movimiento mediante 4 pulsadores que generan interrupciones por GPIO o a través del envío de los caracteres “WASD” desde la computadora hacia la placa mediante recepción UART.
- Envío de instrucciones de juego al encender la placa y envío de estadísticas al finalizar cada partida mediante transmisión UARTm
- Reproducción de un tono a través del DAC al finalizar la partida, lo cual se lleva a cabo transmitiendo datos de memoria al DAC haciendo uso del módulo de DMA.
- Botón de comienzo y reseteo de partida.

Listado de componentes:

- LPC1769 (versión revB)
- Matriz led 8x8 modelo KYX-1088AB
- 8 resistencias 430 Ω
- Potenciómetro 10KΩ
- 5 pulsadores
- Auricular para salida de audio
- Módulo UART TTL a USB tipo A PL2303

Lógica y mecánicas del juego

El código del juego define, en primera instancia, una serie de estructuras y variables globales, las cuales son utilizadas por las funciones que modelan el desarrollo del juego. La descripción de cada una de estas variables y estructuras se describe mediante comentarios:

```

13 #define ANCHO 8
14 #define ALTO 8

28 //Ubicaciones dentro de la matriz 8x8
29 typedef struct {
30     uint8_t x, y;
31 } Point;
32
33 //Direcciones en la que puede moverse la vibora
34 typedef enum {
35     ARRIBA, ABAJO, IZQ, DER
36 } Direction;
37
38 typedef enum {
39     EASY, NORMAL, HARD
40 } Difficulty;
41
42 Point snake[ANCHO * ALTO];    // Arreglo de posiciones ocupadas por la vibora
43 uint8_t snakeLength;          // Cantidad de posiciones ocupadas por la vibora
44 Point apple;                 // Ubicación actual de la manzana a comer
45 Direction direction;         // Dirección actual en la que se mueve la vibora
46 uint8_t appleCounter = 0;     // Cantidad de manzanas ya comidas
47 uint16_t secondsCounter;     // Duración de la partida en segundos
48 Difficulty difficulty;       // Dificultad actual de la partida
49 Bool start = FALSE;          // Bandera de comienzo de la partida inicial
50

```

Imagen 1: Variables globales y estructuras.

main():

Primero, desplaza las muestras de la señal sinusoidal en 6 posiciones mediante un bucle for, la función de estas muestras y el por qué se desplazan se explica mejor en el apartado “**Reproducción de audio mediante DMA y DAC**”. A continuación, llama a las funciones de configuración de UART, botones de movimiento, pines GPIO para control de la matriz LED y ADC. Finalmente, espera hasta que la flag *start* se ponga en 1 para entrar en un bucle while, que se ejecuta durante toda la duración de la partida y llama constantemente a la función render() luego de un *delay(100)* que dura 1,2 milisegundos aproximadamente .

```

84 int main() {
85     // se desplazan las muestras en 6 posiciones
86     // VALUE en el register DACR comprende los bits 15-6
87     for(uint8_t index = 0; index<SAMPLES_AMOUNT; index++) {
88         sinSamples[index] = sinSamples[index]<<6;
89     }
90
91     configUART();
92     helloWorld();
93     configButtons();
94     configGPIO();
95     configADC();
96
97     while(!start){ //Espero que el boton de start levante la flag antes de continuar con el juego
98         delay(300);
99     }
100
101    while (1) { //Una vez configurado e iniciado el juego, renderizo las posiciones de la vibora y la manzana
102        render();
103        delay(100); //SACAR CUENTAS
104    }
105
106    return 0;
107 }
```

Imagen 2: Función main().

initGame():

Setea los parámetros de iniciales de la víbora, llama a las funciones de configuracion del Timer 0 y el SysTick(), le ordena al ADC realizar una única conversión de manera inmediata (con el fin de setear mas adelante la velocidad del juego) y finalmente habilita el Timer0.

```

273 //Inicializa todo lo necesario para una nueva partida
274
275 void initGame() {
276     // Setea la longitud de la vibora y su dirección
277     snakeLength = 3;
278     direction = DER;
279     appleCounter = 0; //Resetea los contadores de manzanas comidas y segundos transcurridos
280     secondsCounter =0;
281
282     snake[0].x = 2; snake[0].y = 4;
283     snake[1].x = 1; snake[1].y = 4;
284     snake[2].x = 0; snake[2].y = 4;
285
286     apple.x=6;
287     apple.y=4;
288
289     configTimer0(); //Timer encargado del tick de movimiento de la vibora
290     ADC_StartCmd(LPC_ADC,ADC_START_NOW); //Hace una unica conversion para obtener la velocidad de juego
291     NVIC_EnableIRQ(ADC_IRQn);
292     configSysTick();
293     TIM_Cmd(LPC_TIM0,ENABLE); //Habilita el timer encargado del tick de movimiento de la vibora
294 }
```

Imagen 3: Función initGame().

moveSnake()

Es responsable del movimiento de la viborita, actualiza la posición de la cabeza de esta basándose en la dirección actual. Chequea que el movimiento sea válido llamando a la función *checkCollisions()*, si lo es, mueve las posiciones en la dirección correspondiente, en caso contrario, es decir, cuando hay una colisión, llama a la función *stopGame()*. Esta función es llamada cada vez que ocurre una interrupción por timer0.

```

297 // Genera la nueva posición de la vibora y si es válida la actualiza en el arreglo snake
298 void moveSnake() {
299     Point newPos = snake[0]; //Copia de la posición actual de la cabeza de la vibora
300     if(direction==ARRIBA) {
301         newPos.y++;
302     } else if(direction==ABAJO) {
303         newPos.y--;
304     } else if(direction==DER) {
305         newPos.x++;
306     } else{           //direction==IZQ
307         newPos.x--;
308     }
309
310     if(!checkCollisions(newPos)){ //Chequeo si estoy haciendo un movimiento válido
311         for(int i=snakeLength-1;i>0;i--) { //Recorro el arreglo en orden inverso
312             snake[i]=snake[i-1];    //Muevo las posiciones de la vibora un lugar dentro del array
313         }
314         snake[0]=newPos; //Guardo la nueva posición de la cabeza de la vibora
315     } else{
316         stopGame();
317     }
318 }
```

Imagen 4: Función moveSnake().

```

519 void TIMER0_IRQHandler() {
520     moveSnake();
521     TIM_ClearIntPending(LPC_TIM0, TIM_MR0_INT);
522 }
```

Imagen 5: Handler timer 0 moveSnake().

checkCollisions():

1. **Choque contra los límites:** Si la nueva posición está fuera de los límites de la matriz, se retorna -1.
2. **Collisión consigo misma:** Si la nueva posición coincide con alguna de las posiciones del cuerpo de la víbora, se retorna -1.
3. **La víbora come una manzana:** Si la nueva posición coincide con la de la manzana, se actualiza el largo de la vborita, la cantidad de manzanas comidas, una nueva manzana es generada y retorna 0.
4. **Movimiento a un espacio vacío:** si ninguna de las situaciones anteriores se da, la función retorna 0, indicando que hay un movimiento válido a un espacio vacío.

```

320 /* Chequea si el proximo movimiento newPos de la vibora contra cuatro situaciones:
321 1) Fuera de los limites -> GameOver: Sonido + StopTotal + Enviar stats
322 2) Choque contra si misma -> GameOver
323 3) Choque con la manzana -> moveSnake + createNewApple + updateLength
324 4) Espacio libre -> moveSnake
325 ---
326     Return: -1 = Game Over // 0 = moveSnake
327 */
328 uint8_t checkCollisions(Point newPos) {
329     //Caso 1: Fuera de los limites
330     if(newPos.x>=ANCHO || newPos.x<0 || newPos.y>=ALTO || newPos.y<0) {
331         return -1;
332     }
333     //Caso 2: Choque consigo misma
334     for (int i = 0; i < snakeLength; i++) {
335         if(newPos.x==snake[i].x && newPos.y==snake[i].y) {
336             return -1;
337         }
338     }
339     //Caso 3: Choque con una manzana
340     if (newPos.x==apple.x && newPos.y==apple.y) {
341         snakeLength++;
342         appleCounter++;
343         createNewApple();
344         return 0;
345     }
346     //Caso 4: Movimiento válido a espacio vacío
347     return 0;
348 }
```

Imagen 6: Función checkCollisions().

updateDirection():

Actualiza la dirección de la víbora si el nuevo valor no es igual al que ya tenía y tampoco es igual a la dirección que se debe evitar (Es decir, si la vborita, por

ejemplo, se está desplazando hacia la derecha, no puede moverse a la izquierda y viceversa).

```

350/* Cuando el pulsador envia su dirección correspondiente, chequea si es valido
351  y si es así actualiza la dirección actual de la víbora
352 */
353void updateDirection(Direction new, Direction avoid) {
354    if(direction!= new && direction!=avoid){
355        direction=new;
356    }
357}

```

Imagen 7: Función updateDirection().

createNewApple():

Genera una nueva manzana llamando a getRandomPair(), asegurándose que la posición de esta no coincida con alguna de las posiciones que ya está ocupando la víbora.

```

359/* Genera posición random para nueva manzana
360  - Chequea si la posición está ocupada por la víbora
361  - Update de la posición al elemento "apple"
362 */
363void createNewApple() {
364    Point newApple;
365    uint8_t flag = 1;
366    while(flag!=0){
367        flag = 0;
368        getRandomPair(&newApple.x,&newApple.y);
369        for(int i=0;i<snakeLength;i++){ //verifico la posición de newApple contra todas las de la víbora
370            if(snake[i].x==newApple.x && snake[i].y==newApple.y){
371                flag++; //Si encuentra una coincidencia, levanto la bandera
372            }
373        }
374    }
375    apple=newApple; //Guardo la nueva posición
376}

```

Imagen 8: Función createNewApple().

getRandomPair()

Genera un par de valores random entre 0-7 (correspondiente a una posición dentro de la matriz LED), utilizando como seed el valor actual del SysTick,el cual interrumpe cada 1 milisegundo.

```

378 //Usa el value de Systick para generar dos valores en el rango [0:7] que guarda en a y b
379void getRandomPair(uint8_t* a, uint8_t* b){
380    volatile uint32_t seed = SysTick->VAL;
381    seed ^= (seed << 13);
382
383    *a = (seed & 0x07);
384    *b = ((seed >> 3)& 0x07);
385}

```

Imagen 9: Función getRandomPair().

helloWorld() y sendStats():

Estas funciones utilizan el módulo UART para enviar información a la PC. `helloWorld()` envía el saludo inicial y las instrucciones de juego cuando se prende el circuito por primera vez (no se adjunta imagen debido a la longitud horizontal de la misma, pero se puede ver en detalle a partir de la línea 421). `sendStats()`, por otra parte, se encarga de enviar las estadísticas de la partida al finalizar la misma:

- ID de partida
- Dificultad seleccionada
- Duración en segundos, contados mediante el `SysTick()`, el cual interrumpe cada 1 milisegundo
- Cantidad de manzanas comidas

Para poder mostrar números enteros en consola se hace uso de la función `uint16_to_uintArray`, la cual convierte un número entero en un array

```

508 //Lleva la cuenta de los segundos de la partida
509 void SysTick_Handler(){
510     static uint16_t millisCount = 0;
511     millisCount++;
512
513     if(millisCount >= 1000){
514         secondsCounter++;
515         millisCount = 0;
516     }
517 }
```

Imagen 10: Handler SysTick y contador de milisegundos

```

387 //Se encarga de enviar las estadísticas de la partida a la PC
388 void sendStats() {
389     static uint8_t gameCounter = 0;      //Contador de partidas
390     gameCounter++;
391     uint8_t numbers[4]; //Buffer para el array de dígitos
392     uint8_t digitos; //Auxiliar para contar la cantidad de dígitos en el buffer
393
394     uint8_t data0[] = "\n\rChan chan chan...Se terminó el juego mi loco! Acá van un par de estadísticas:\n\r";
395     UART_Send(LPC_UART1,data0, sizeof(data0), BLOCKING);
396
397     UART_Send(LPC_UART1,(uint8_t*)" ID de partida: ",17, BLOCKING);
398     digitos=uint16_to_uint8Array(gameCounter, numbers);
399     UART_Send(LPC_UART1,(uint8_t *)numbers, digitos, BLOCKING);
400
401     UART_Send(LPC_UART1,(uint8_t*)" \n\r Dificultad seleccionada: ",29, BLOCKING);
402     if(difficulty==EASY) {
403         UART_Send(LPC_UART1,(uint8_t *)"FACIL",5,BLOCKING);
404     } else if(difficulty==NORMAL) {
405         UART_Send(LPC_UART1,(uint8_t *)"NORMAL",6,BLOCKING);
406     } else{
407         UART_Send(LPC_UART1,(uint8_t *)"DIFICIL",7,BLOCKING);
408     }
409     UART_Send(LPC_UART1,(uint8_t*)" \n\r Duración de la partida en segundos: ",41, BLOCKING);
410     digitos=uint16_to_uint8Array(secondsCounter, numbers);
411     UART_Send(LPC_UART1,(uint8_t *)numbers, digitos, BLOCKING);
412
413     UART_Send(LPC_UART1,(uint8_t*)" \n\r Manzanas comidas: ",22, BLOCKING);
414     digitos=uint16_to_uint8Array(appleCounter, numbers);
415     UART_Send(LPC_UART1,(uint8_t *)numbers, digitos, BLOCKING);
416
417     UART_Send(LPC_UART1,(uint8_t *)" \n\r",2,BLOCKING);
418 }

```

Imagen 11: Función sendStats().

```

462 //Convierte un entero de 16bits a un string de dígitos en ASCII
463 uint8_t uint16_to_uint8Array(uint16_t value, uint8_t *result) {
464     // Buffer size based on the maximum number of digits in a uint16_t (5 digits)
465     uint8_t buffer[5];
466     // Initialize index
467     uint8_t index = 0;
468
469     // Handle the case when the value is 0 separately
470     if (value == 0) {
471         buffer[index++] = '0';
472     } else {
473         // Extract digits in reverse order
474         while (value > 0) {
475             buffer[index++] = '0' + (value % 10);
476             value /= 10;
477         }
478     }
479     // Reverse the buffer to get the correct order
480     for (int8_t i = 0; i < index; ++i) {
481         result[i] = buffer[index - 1 - i];
482     }
483     // Null-terminate the result
484     result[index] = '\0';
485     return index;
486 }

```

Imagen 12: Función uint16_to_uint8Array().

render():

Es responsable de actualizar la matriz para mostrar el estado actual de la viborita y la manzana. La matriz LED está representada por 2 arrays:

- X representa las columnas. Cada elemento de este arreglo indica que pines del puerto 2 (especificados en el apartado “Matriz Led 8x8”) que pin se debe poner en alto, con el resto en 0.
- Y representa las filas. Cada elemento de este arreglo indica que pin del puerto 0 (especificados en el apartado de diagramas de conexiones) se debe poner en bajo, dejando el resto en 1.

Cuando estos arreglos se recorren de izquierda a derecha, el desplazamiento a lo largo de la matriz es como se muestra en la siguiente imagen

```
static uint16_t X[8]={0x0020,0x0001,0x0002,0x0008,0x0004,0x0010,0x0040,0x0080};

static uint16_t Y[8]={0x0DF0,0x07F0,0x0EF0,0x0BF0,0x0FE0,0x0F70,0x0FD0,0x0FB0};
```

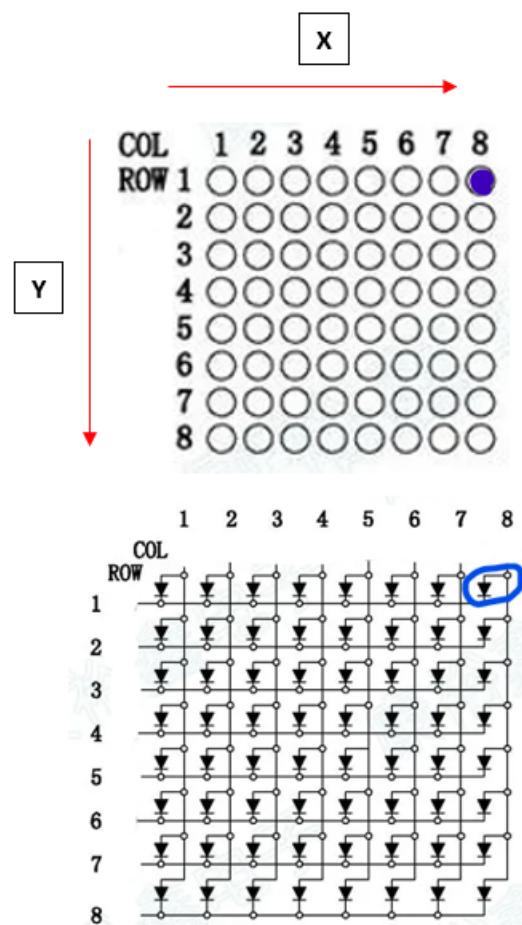


Imagen 13: Arreglos de posiciones X e Y.

Por lo tanto, si se quiere por ejemplo encender el Led ubicado en la columna 8 y fila 1, se setea el puerto 2 en 0x0080 y el puerto 0 en 0x0DF0.

Mediante una variable estática, se prende un Led particular correspondiente a una de las posiciones de la serpiente o a la posición de la manzana en cada llamado a esta función. **Los llamados se hacen a una velocidad lo suficientemente alta como para que al ojo humano interprete que hay varios leds prendidos al mismo tiempo** (ver función *delay()* en main).

```

430 //Chequea y envía los leds a encender a la matriz
431 void render() {
432
433     static uint16_t X[8]={0x0020,0x0001,0x0002,0x0008,0x0004,0x0010,0x0040,0x0080};
434
435     static uint16_t Y[8]={0x0DF0,0x07F0,0x0EF0,0x0BF0,0x0FE0,0x0F70,0x0FD0,0x0FB0};
436
437     static int i = 0;
438     if(i>=(snakeLength+1)){
439         i=0;
440     }
441
442     if(!i){      //Renderizar posición de la manzana
443         LPC_GPIO2->FIOPINL = X[apple.x];
444         LPC_GPIO0->FIOPINL = Y[apple.y];
445     } else{
446         LPC_GPIO2->FIOPINL = X[snake[i-1].x];
447         LPC_GPIO0->FIOPINL = Y[snake[i-1].y];
448     }
449     i++;
450 }
```

Imagen 14: Función render().

stopGame():

Deshabilita el timer 0 para detener el movimiento de la viberita y el SysTick para el conteo de los segundos. Se envían las estadísticas de la partida mediante *sendStats()* y finalmente se llama a las funciones de configuración del DAC, junto con el canal DMA asociado a este periférico (para que se reproduzca el sonido de fin de juego).

```

454 void stopGame() {
455     TIM_Cmd(LPC_TIM0,DISABLE);
456     SYSTICK_Cmd(DISABLE);
457     sendStats();
458     configDAC();
459     configDMA_DAC_Channel();
460 }
```

Imagen 15: Función stopGame()

Matriz Led 8x8 y pines que la controlan

El juego se muestra sobre una matriz led 8x8 de leds azules de tipo ánodo común. El diagrama de pines se muestra a continuación:

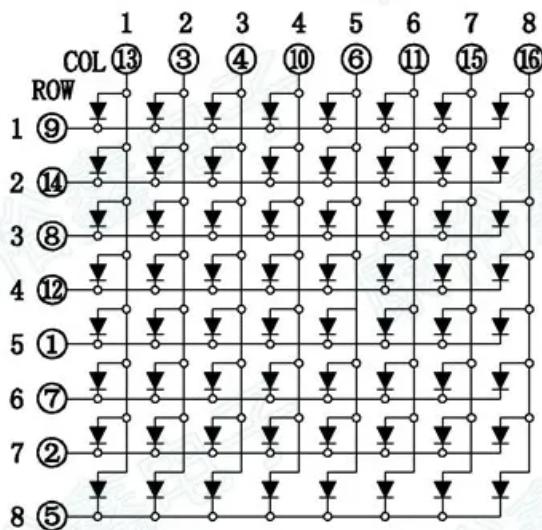


Imagen 16: Matriz LED modelo KYX-1088AB

Para controlar la matriz, se usaron los pines P2.0 a P2.7 para controlar los pines que deben setearse en alto (ver imagen X, aquellos que “seleccionan” una columna en particular). En el caso de los pines que deben ponerse en bajo para acceder a una fila en particular, se usaron los pines P0.4 a P0.11.

Pin Matriz LED que se pone en alto	16	15	13	11	10	6	4	3
Pin de la placa que lo controla	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0

Pin Matriz LED que se pone en bajo	14	12	9	8	7	5	2	1
Pin de la placa que lo controla	P0.11	P0.10	P0.9	P0.8	P0.7	P0.6	P0.5	P0.4

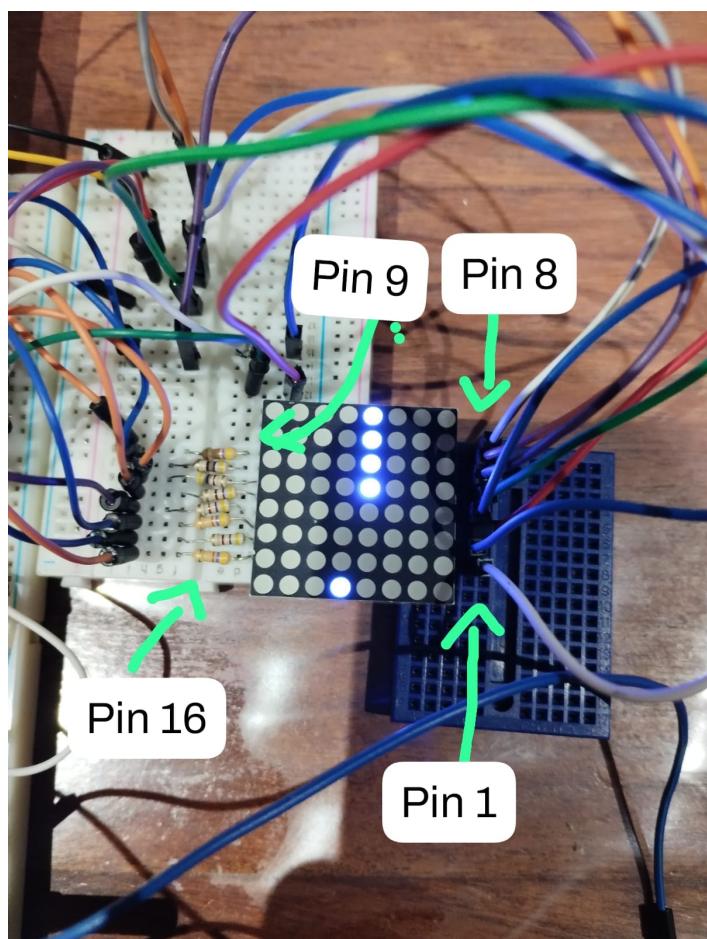


Imagen 17: Matriz LED, se colocan resistencias de 430 ohm para limitar la corriente que llega a los leds

Botones de movimiento y Start/Restart(GPIO)

Para el control de la viberita mediante pulsadores, se configuran los pines P0.0-P0.7 para generar interrupciones (GPIO) cuando ocurre un flanco ascendente en cualquiera de ellos, habilitando también las resistencias pull down integradas para evitar problemas de rebote ligados a los pulsadores. En la rutina de interrupción de cada uno de los pines se modifica la dirección de la viberita según corresponda.

El botón de restart se implementa usando el P0.22, al igual que para los controles de movimiento, se habilita también la interrupción por flanco descendente y la resistencia de pull down. Dentro de la rutina de interrupción del botón, se pone pone en alto la flag de *start* en caso de que no lo esté (ver función *main()*) y llama a la función *initGame()* poniendo en marcha el juego.

```

132 void configButtons() {
133     //Se habilita resistencias de pull down en los pines P0.0 a P0.3
134     // P0.0---->ARRIBA
135     // P0.1---->DERECHA
136     // P0.2---->IZQUIERDA
137     // P0.3---->ABAJO
138     LPC_PINCON->PINMODE0 |= (3<<0);
139     LPC_PINCON->PINMODE0 |= (3<<2);
140     LPC_PINCON->PINMODE0 |= (3<<4);
141     LPC_PINCON->PINMODE0 |= (3<<6);
142
143     // Se habilita resistencia de pull down en P0.22
144     // P0.22---->START/RESTART
145     LPC_PINCON->PINMODE1 |= (3<<12);
146
147     // Se habilita interrupción por flanco de subida en todos los botones
148     LPC_GPIODINT->IO0IntEnR |=0x0000000F;
149     LPC_GPIODINT->IO0IntEnR |= (1<<22);
150
151     // Se limpian banderas de interrupción
152     LPC_GPIODINT->IO0IntClr |=0x0000000F;
153     LPC_GPIODINT->IO0IntClr |= (1<<22);
154
155     NVIC_EnableIRQ(EINT3_IRQn);
156 }
```

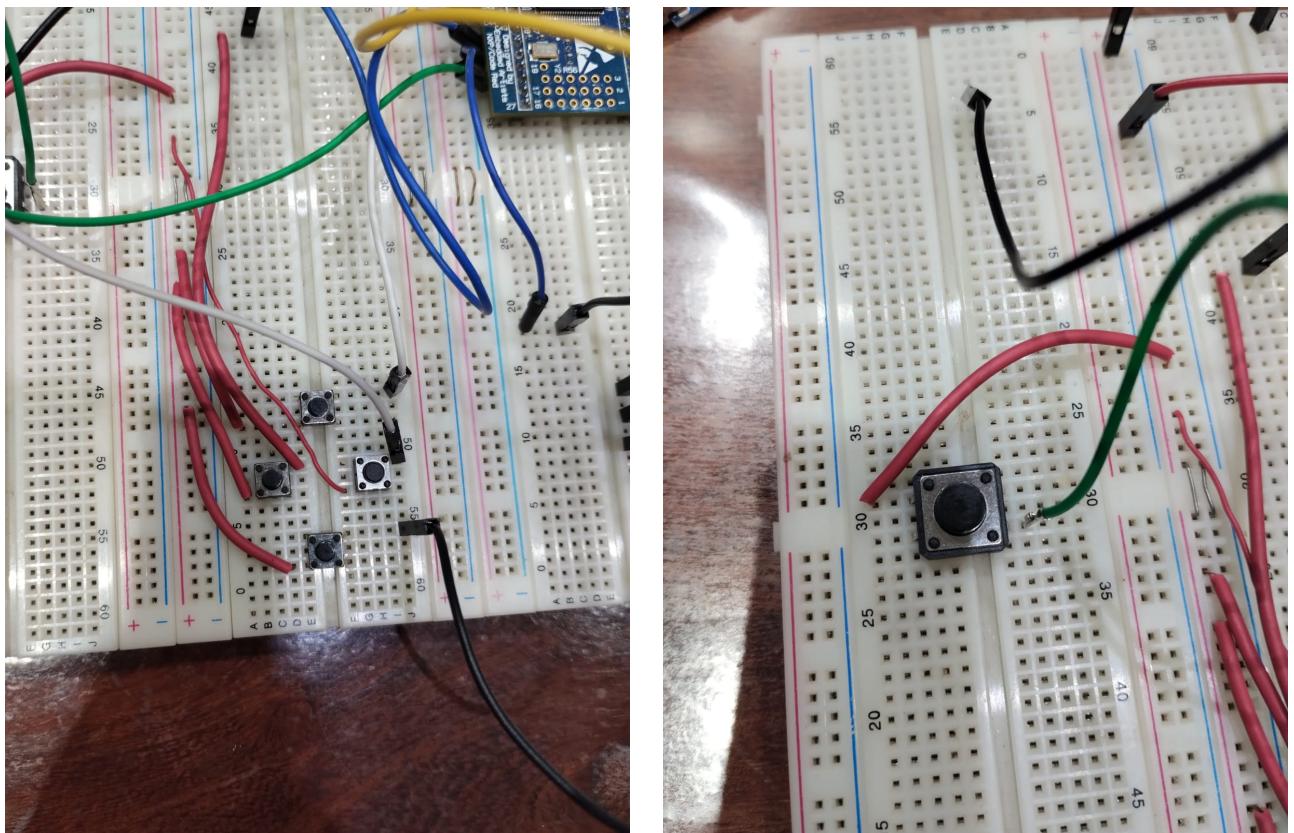
Imagen 18: Función de configuración de los botones de movimiento y start/restart

```

521 void EINT3_IRQHandler() {
522     //ARRIBA
523     if((LPC_GPIOINT->IO0IntStatR) &(1<<0)) {
524         updateDirection(DER,IZQ);
525         LPC_GPIOINT->IO0IntClr |=(1<<0);
526     }
527     //DERECHA
528     else if((LPC_GPIOINT->IO0IntStatR) &(1<<1)) {
529         updateDirection(ARRIBA,ABAJO);
530         LPC_GPIOINT->IO0IntClr |=(1<<1);
531     }
532     //IZQUIERDA
533     else if((LPC_GPIOINT->IO0IntStatR) &(1<<2)) {
534         updateDirection(ABAJO,ARRIBA);
535         LPC_GPIOINT->IO0IntClr |=(1<<2);
536     }
537     //ABAJO
538     else if((LPC_GPIOINT->IO0IntStatR) &(1<<3)) {
539         updateDirection(IZQ,DER);
540         LPC_GPIOINT->IO0IntClr |=(1<<3);
541     }
542     // BOTON DE START/RESTART
543     else{
544         if(!start){
545             start = TRUE;
546         }
547         initGame();
548         LPC_GPIOINT->IO0IntClr |=(1<<22);
549     }
550 }

```

Imagen 19: Matriz LED modelo KYX-1088AB



Imagenes 20 y 21: botones de movimiento y botón de start/restart

Regulador de velocidad (ADC y potenciómetro)

Se configura el canal 0 del ADC para realizar una única conversión e interrumpir una vez completada (ver función initGame() en la sección de “Lógica y mecánicas del juego”). El valor muestreado se usa para decidir la velocidad del juego antes de habilitar el timer 0, configurando el valor de match correspondiente. Si el valor muestreado es mayor a 3500, la dificultad seleccionada es difícil, si es mayor a 1000 y menor a 3500 es normal y si es menor a 1000 el juego se pone en modo fácil.

```

255 void configADC() {
256     PINSEL_CFG_Type PinCfg;
257     PinCfg.Funcnum = 1;
258     PinCfg.OpenDrain = 0;
259     PinCfg.Pinmode = 0;           //Sin pull-up ni pull-down
260     PinCfg.Pinnum = 23;
261     PinCfg.Portnum = 0;
262     PINSEL_ConfigPin(&PinCfg); //P0.23 como AD0.0
263
264     ADC_Init(LPC_ADC, 200000);      //Frec. de muestreo = 200kHz
265     ADC_IntConfig(LPC_ADC, ADC_ADINTEN0, ENABLE); //Habilito interrupción canal 0
266     ADC_ChannelCmd(LPC_ADC, ADC_CHANNEL_0, ENABLE); //Habilito canal 0
267 }
```

Imagen 22: Configuración canal 0 ADC

```

16 #define HARD_MAX    3500
17 #define NORMAL_MAX  1000
18
19 #define TIMER_EASY   1300
20 #define TIMER_NORMAL 800
21 #define TIMER_HARD   400
22
```

Imagen 23: Constantes

```

555 void ADC_IRQHandler() {
556     IO uint32_t adcValue = 0; //Uso variable local, no hay necesidad de tenerla como global
557     if (ADC_ChannelGetStatus(LPC_ADC, ADC_CHANNEL_0, ADC_DATA_DONE)) { //Leo el valor de connversión en el canal 0
558         adcValue = ADC_ChannelGetData(LPC_ADC, ADC_CHANNEL_0);
559         NVIC_DisableIRQ(ADC_IRQn);
560     }
561
562     //A menor valor en la medición, mayor es la resistencia del potenciómetro
563     //Escala de medición del ADC: 0--(Zona difícil)--HARD_MAX--(Zona normal)--NORMAL_MAX--(Zona fácil)--4095
564
565     if(adcValue>HARD_MAX){          //El potenciómetro está cerca de su valor mínimo
566         TIM_UpdateMatchValue(LPC_TIM0,0,TIMER_HARD);
567         difficulty = HARD;
568     } else if(adcValue>NORMAL_MAX){ //El potenciómetro está en un valor intermedio
569         TIM_UpdateMatchValue(LPC_TIM0,0,TIMER_NORMAL);
570         difficulty = NORMAL;
571     } else{                      //El potenciómetro está cerca de su valor maximo
572         TIM_UpdateMatchValue(LPC_TIM0,0,TIMER_EASY);
573         difficulty = EASY;
574     }
575
576     LPC_ADC->ADGDR &= LPC_ADC->ADGDR;
577 }
```

Imagen 24: Elección de dificultad y configuración de valor de match del timer 0.

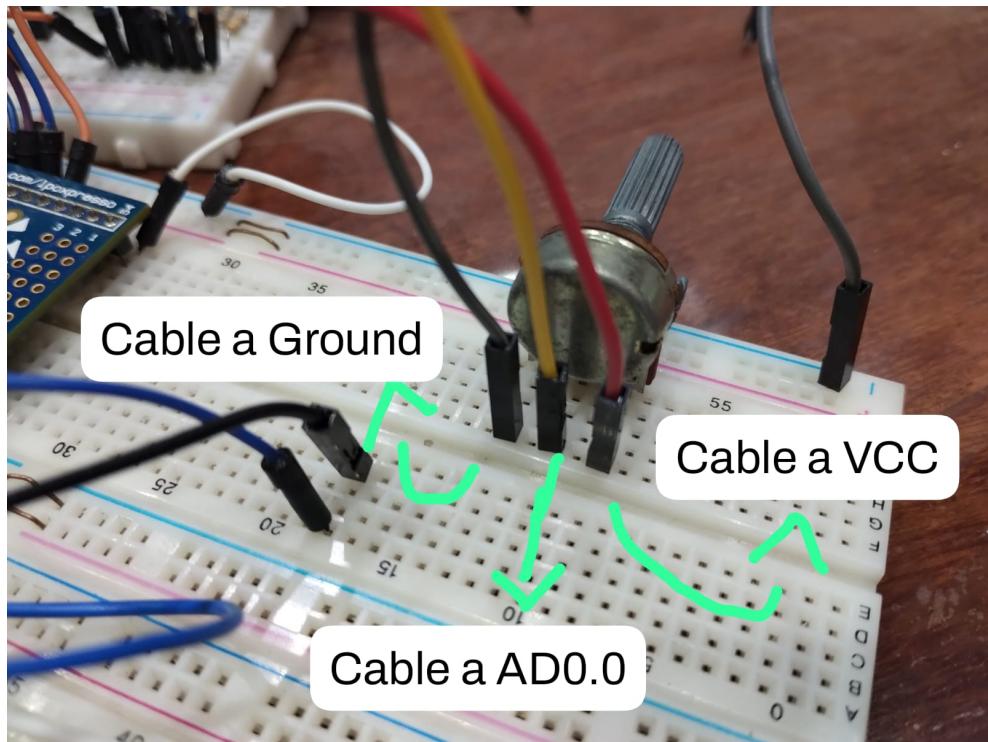


Imagen 25: Conexiones del potenciómetro

Reproducción de audio mediante DMA y DAC

Tal como se podía observar en la función `stopGame()`, al finalizar la partida se reproduce una onda sinusoidal a través del DAC, transfiriendo las muestras que se encuentran en el array hacia este periférico por el canal DMA 0. Cada una de las muestras de la señal sinusoidal se desplazan 6 lugares en la función `main()`, debido a que el valor de 10 bits a cargar en el DAC para convertirse en salida analógica se encuentra comprendido entre los bits [15-6] del registro DACR. El sonido es reproducido mediante un auricular y solo deja de transmitirse cuando se inicia una nueva partida.

Cálculos:

Dado que la onda que se emite es de 400Hz y se tiene un total de 60 muestras, el tiempo de time out del DAC (tiempo que transcurre entre solicitudes de transferencia de dato por DMA) se calcula del siguiente modo:

$$60 \text{ Muestras} = \frac{1}{400\text{Hz}}$$

$$1 \text{ Muestra} = X$$

$$X = \frac{1}{\frac{400\text{Hz}}{60}} = 4,166 * 10^{-5}\text{s} = \text{tiempo entre muestras}$$

Para generar, dicho tiempo, se debe tener en cuenta que el clock de CPU es de 100MHz y el PCLK_DAC es igual al valor mencionado dividido 4, entonces:

$$\text{time out counter value} = 1 = \frac{1}{25\text{MHz}}$$

$$\text{time out counter value} = \text{tmp} = 4,166 * 10^{-5}$$

$$\text{tmp} = \frac{(4,166 * 10^{-5})}{\frac{1}{25\text{MHz}}} \approx 1042$$

Configuración:

La configuración completa del DAC y el canal DMA se lleva a cabo con las funciones configDAC() y configDMA_DAC_Channel().

```
23 #define SAMPLES_AMOUNT 60
24 #define SINE_FREQ_IN_HZ 400
25 #define PCLK_DAC_IN_MHZ 25
```

Imagen 26: Constantes para el cálculo del DAC time out counter value.

```
78 uint32_t sinSamples[SAMPLES_AMOUNT] = {
79     511, 564, 617, 669, 719, 767, 812, 853, 891, 925, 954, 978, 997, 1011, 1020, 1023,
80     1020, 1011, 997, 978, 954, 925, 891, 853, 812, 767, 719, 669, 617, 564, 511, 458,
81     405, 353, 303, 255, 210, 169, 131, 97, 68, 44, 25, 11, 2, 0, 2, 11, 25, 44, 68,
82     97, 131, 169, 210, 255, 303, 353, 405, 458};
```

Imagen 27: Muestras de la onda sinusoidal

```

200 void configDAC() {
201     //Configuración de P0.26 como salida analógica del DAC
202     PINSEL_CFG_Type pinCfg;
203     pinCfg.Funcnum = 2;
204     pinCfg.OpenDrain = 0;
205     pinCfg.Pinmode = 0;
206     pinCfg.Portnum = 0;
207     pinCfg.Pinnum = 26;
208     PINSEL_ConfigPin(&pinCfg);
209
210     DAC_CONVERTER_CFG_Type dacCfg;
211     dacCfg.CNT_ENA = SET;
212     dacCfg.DMA_ENA = SET;
213     DAC_Init(LPC_DAC);
214
215     // configuración tiempo de time out DAC
216     uint32_t tmp;
217     tmp = (PCLK_DAC_IN_MHZ * 1000000)/(SINE_FREQ_IN_HZ * SAMPLES_AMOUNT);
218     DAC_SetDMATimeOut(LPC_DAC, tmp);
219     DAC_ConfigDACConverterControl(LPC_DAC, &dacCfg);
220 }

```

Imagen 28: Configuración del pin de salida y parámetros del DAC

```

222 void configDMA_DAC_Channel(){
223     /*-----Configuración linked list-----*/
224     //source width 32 bits
225     //destination width 32 bits
226     //source address se incrementa en cada transmisión
227     //destination address (DAC) se mantiene fija
228     GPDMA_LLI_Type LLI1;
229     LLI1.SrcAddr = (uint32_t) sinSamples;
230     LLI1.DstAddr = (uint32_t) &LPC_DAC->DACR;
231     LLI1.NextLLI = (uint32_t) &LLI1;
232     LLI1.Control = SAMPLES_AMOUNT| (1<<19) | (1<<22) | (1<<26);
233
234     GPDMA_Init();
235
236     // configuracion y habilitacion del Canal 0 de DMA
237     GPDMA_Channel_CFG_Type GPDMACfg;
238     GPDMACfg.ChannelNum = 0;
239     GPDMACfg.SrcMemAddr = (uint32_t)sinSamples;
240     GPDMACfg.DstMemAddr = 0;
241     GPDMACfg.TransferSize = SAMPLES_AMOUNT;
242     GPDMACfg.TransferWidth = 0;
243     GPDMACfg.TransferType = GPDMA_TRANSFERTYPE_M2P;
244     GPDMACfg.SrcConn = 0;
245     GPDMACfg.DstConn = GPDMA_CONN_DAC;
246     GPDMACfg.DMALLI = (uint32_t)&LLI1;
247     GPDMA_Setup(&GPDMACfg);
248     GPDMA_ChannelCmd(0, ENABLE);
249 }

```

Imagen 29:Configuración del canal DMA 0

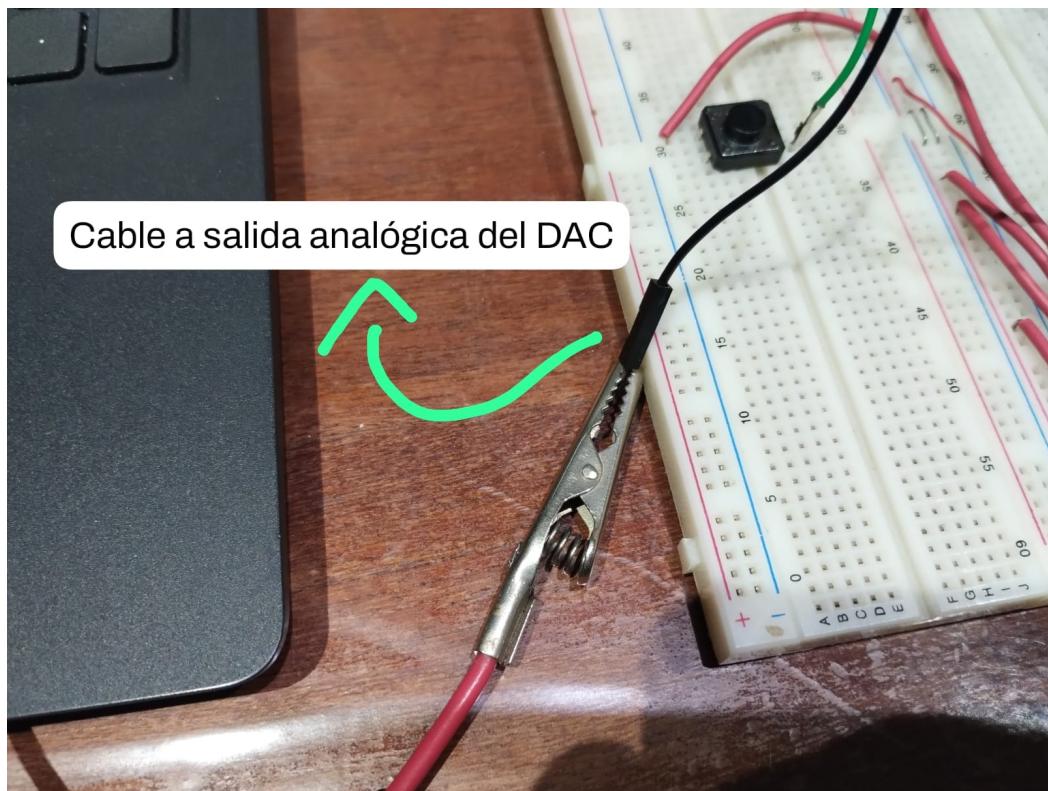


Imagen 30 : Salida DAC

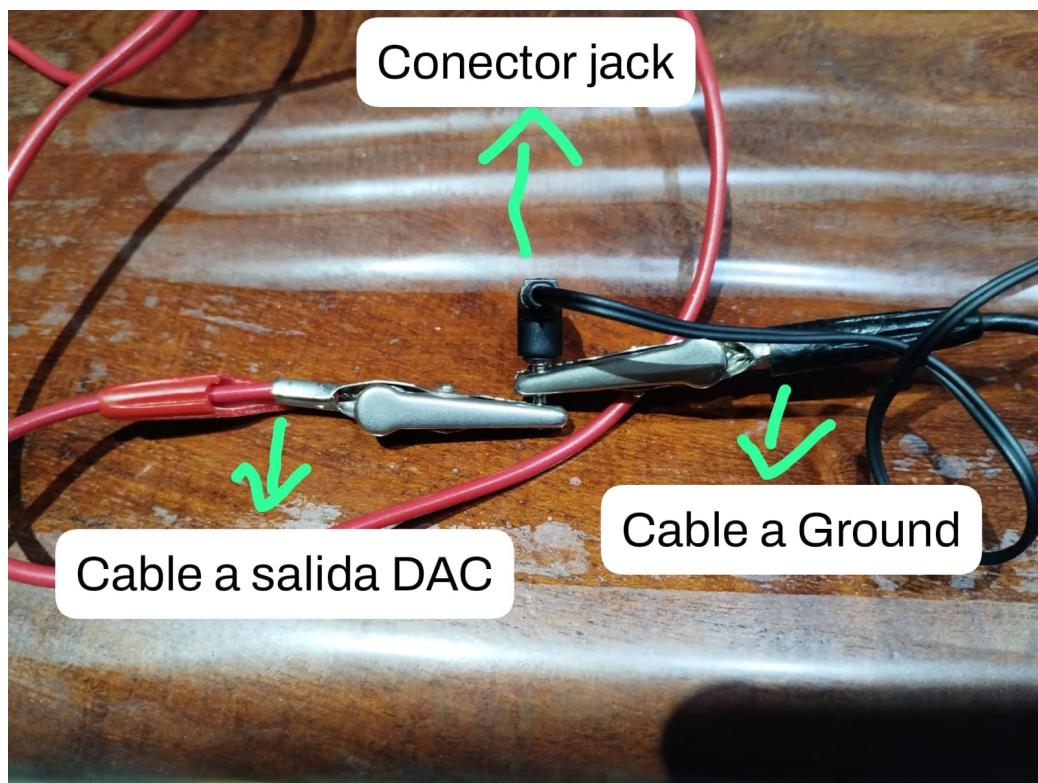


Imagen 31: Conexiones a salida DAC y ground en conector Jack del auricular

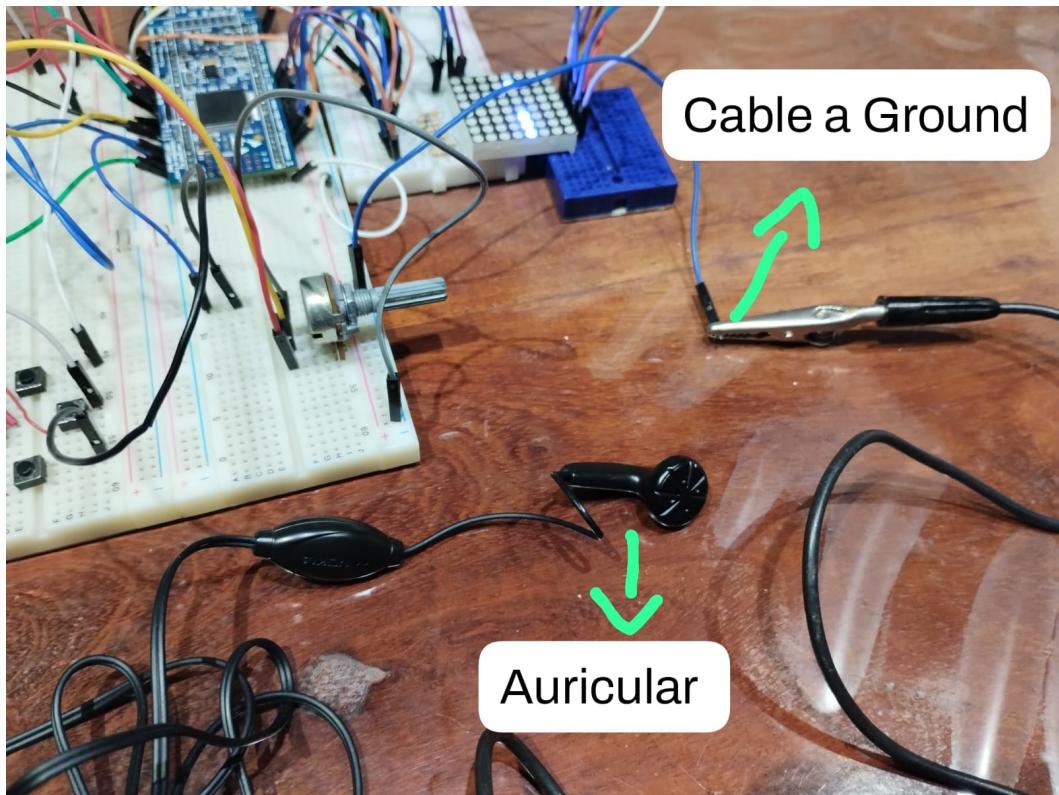


Imagen 32: Auricular y conexión a ground

Envío de estadísticas y control de movimiento por UART 1

Con la ayuda de un conversor UART a USB tipo A, el módulo UART1 se usa, por una parte, para transmitir estadísticas y las instrucciones de juego hacia la PC mediante las funciones `HelloWorld()` y `sendStats()`, dichas funciones se describieron con detalle en la sección “**Lógica y mecánicas del juego**”.

La segunda tarea de este periférico es la de recibir datos de la computadora, concretamente los caracteres “WASD”, permitiendo al jugador optar por 2 formas de control de movimiento de la viborita. Cada vez que se ingresa un carácter por consola, se genera una interrupción en cuyo handler se identifica el carácter ASCII recibido llamando a la función `ASCIItoDirection()`, en caso de que no sea válido, el curso de la viborita no se altera. La configuración del mismo se realiza con la función `configUART()`.

```

171 void configUART() {
172     //Configuro los pines Rx y Tx
173     PINSEL_CFG_Type PinCfg;
174     PinCfg.Funcnum = 1;
175     PinCfg.OpenDrain = 0;
176     PinCfg.Pinmode = 0;
177     PinCfg.Pinnum = 15;
178     PinCfg.Portnum = 0;
179     PINSEL_ConfigPin(&PinCfg); //P0.15
180     PinCfg.Pinnum = 16;
181     PINSEL_ConfigPin(&PinCfg); //P0.16
182
183     UART_CFG_Type UARTConfigStruct;
184     UART_FIFO_CFG_Type UARTFIFOConfigStruct;
185     UART_ConfigStructInit(&UARTConfigStruct); //Usamos la configuración por defecto
186     UART_Init(LPC_UART1, &UARTConfigStruct);
187
188     UART_FIFOConfigStructInit(&UARTFIFOConfigStruct);
189     UART_FIFOConfig(LPC_UART1, &UARTFIFOConfigStruct);
190
191     // Habilita interrupción por el RX del UART
192     UART_IntConfig(LPC_UART1, UART_INTCFG_RBR, ENABLE);
193     // Habilita interrupción por el estado de la linea UART
194     UART_IntConfig(LPC_UART1, UART_INTCFG_RLS, ENABLE);
195
196     UART_TxCmd(LPC_UART1, ENABLE); //Habilitamos la transmisión
197     NVIC_EnableIRQ(UART1_IRQn);
198 }
```

Imagen 33: Configuración del módulo UART1

```

575 void UART1_IRQHandler(void) {
576     uint8_t data[1] = "";
577
578     UART_Receive(LPC_UART1, data, sizeof(data), NONE_BLOCKING);
579     ASCIIItodirection(data[0]);
580 }
```

Imagen 34: Handler UART1

```

488 void ASCIIItodirection(uint8_t value){
489     if(value=='w'){
490         updateDirection(DER,IZQ);
491     } else if(value=='s'){
492         updateDirection(IZQ,DER);
493     } else if(value=='a'){
494         updateDirection(ABAJO,ARRIBA);
495     } else if(value=='d'){
496         updateDirection(ARRIBA,ABAJO);
497     }
498 }
```

Imagen 35: Función *ASCCItodirection()*

```

COM3 - PuTTY

Bueeenas! Gracias por jugar nuestro juego, acá te paso un par de tips sobre como funciona todo:
- Para iniciar la partida apretá el botón de Start/Restart
- Antes de iniciar cada partida vas a poder elejir la dificultad del juego con nuestro selector de velocidad
- Las reglas son bien simples: usá los botones de movimiento para comer todas las manzanas posibles sin chocarte con las paredes o tu propia cola
- Cuando pierdas (no te preocupes, en algún momento todos inevitablemente perdemos) te vamos a pasar algunas estadísticas y reproducir un sonido
- Pero eso no es todo! Queres seguir jugando? Simplemente presioná el botón de Start/Restart y probá tus habilidades de vuelta!!

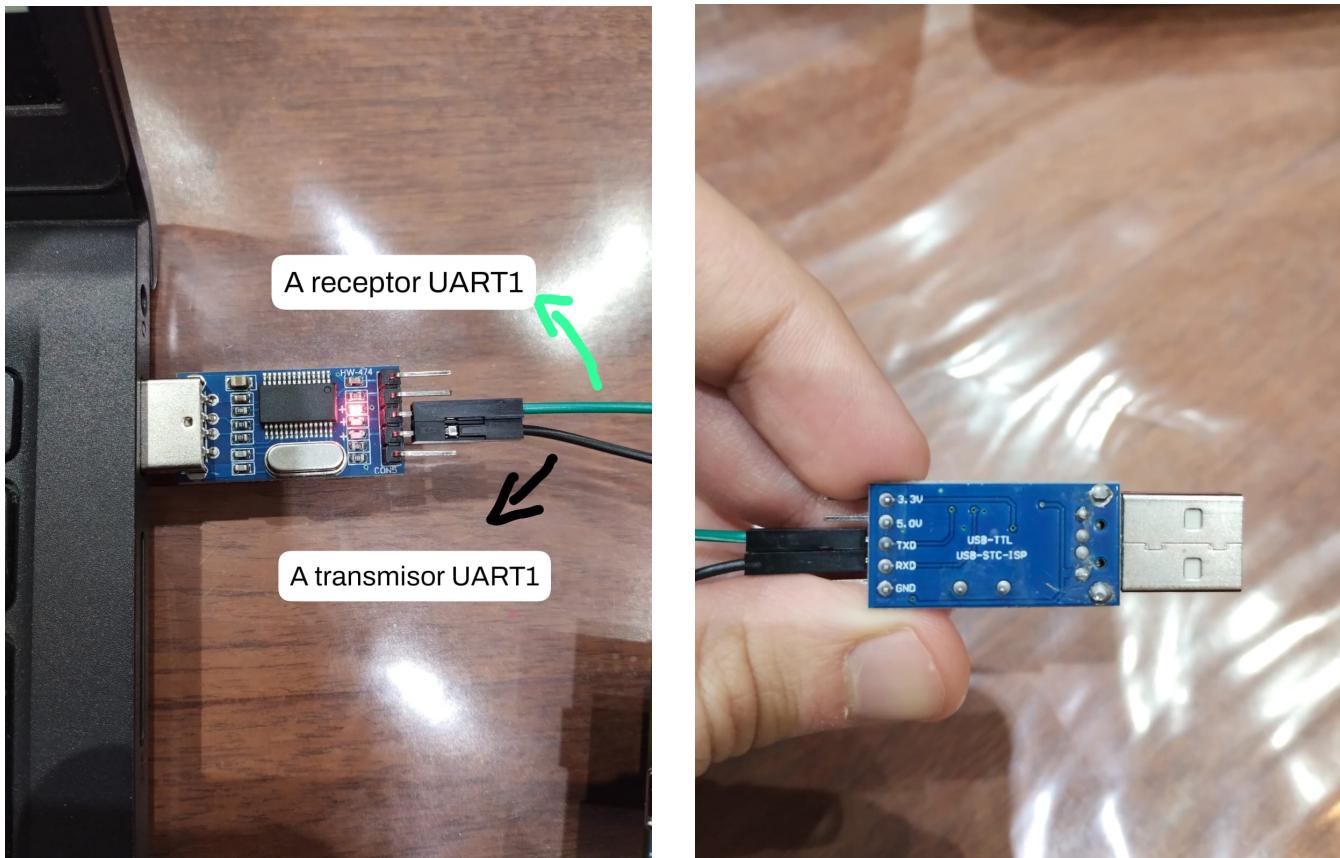
Chan chan chan...Se terminó el juego mi loco! Acá van un par de estadisticas:
ID de partida: 1
Dificultad seleccionada: FACIL
Duración de la partida en segundos: 33
Manzanas comidas: 4

Chan chan chan...Se terminó el juego mi loco! Acá van un par de estadisticas:
ID de partida: 2
Dificultad seleccionada: NORMAL
Duración de la partida en segundos: 32
Manzanas comidas: 7

Chan chan chan...Se terminó el juego mi loco! Acá van un par de estadisticas:
ID de partida: 3
Dificultad seleccionada: DIFICIL
Duración de la partida en segundos: 2
Manzanas comidas: 1

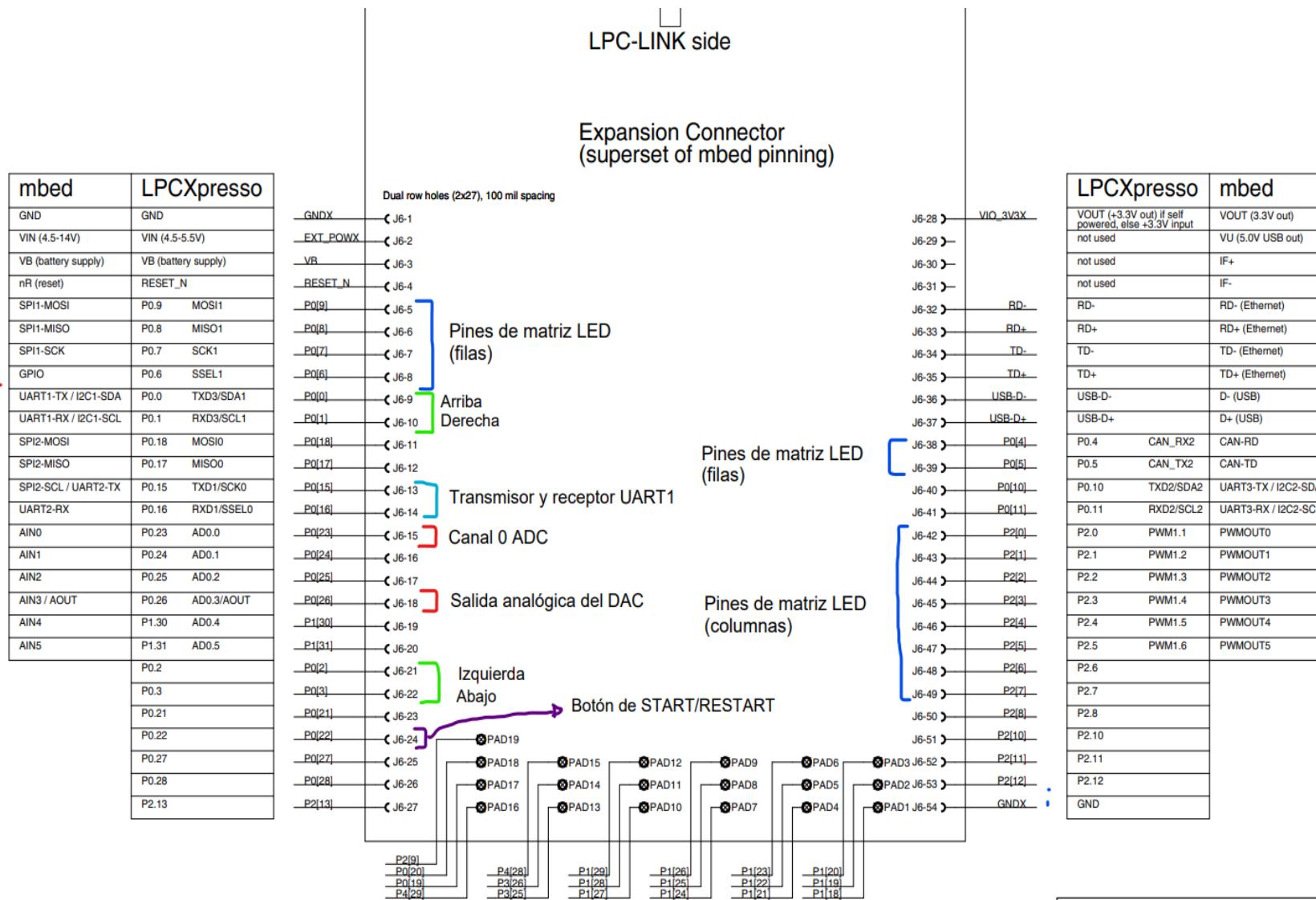
```

Imagen 36: Captura de información transmitida desde la placa a PC, mensaje inicial con las instrucciones de juego y resultados de 3 partidas en donde se varía la dificultad seleccionada.



Imágenes 37 y 38 : vista superior e inferior módulo UART a USB tipo A

Diagrama de conexiones en datasheet



Anexos

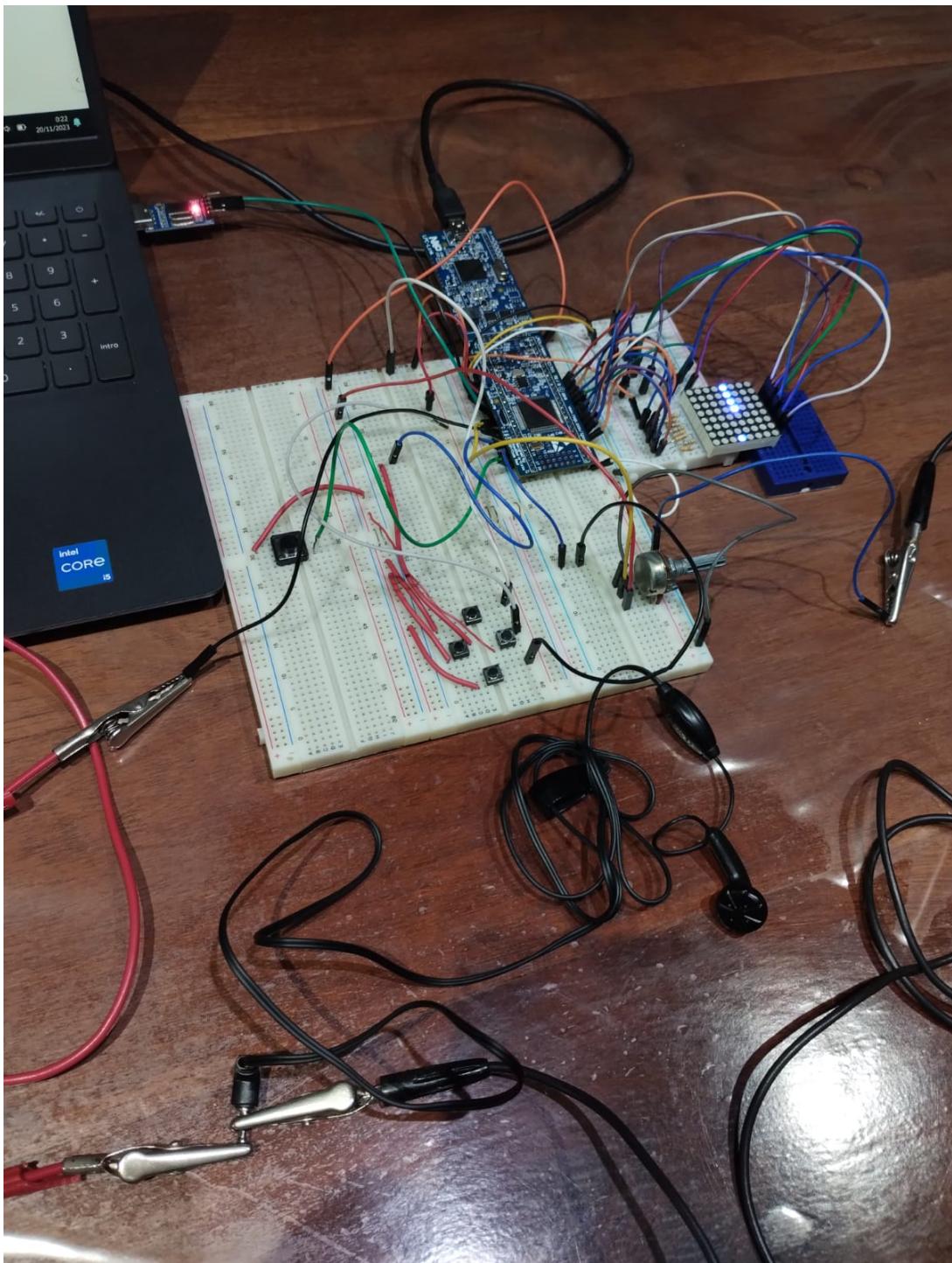


Imagen 39 : Circuito completo

-Datasheet matriz Led: <https://es.aliexpress.com/item/32594255205.html>

-módulo Uart ttl a USB tipo A:

<https://www.todomicro.com.ar/optoacopladores-conversores-y-adaptadores-rs232-rs485-rs422-uart/84-usb-20-a-uart-ttl-5-pines-5v.html>

