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

Contenido

Overview

The following project implemented the classic Snake game on a LPC1769 board (revB version), which contains an ARM Cortex-M3 processor. The game is displayed on an 8x8 LED matrix and has the following features:

● Speed Regulation: Game speed is controled through a potentiometer and the ADC module.

● Motion Control: Using either 4 physical buttons that trigger GPIO interruptions or through the "WASD" characters received via UART communication.

● Statistics Transmission: Statistics are sent through UART transmission at the end of each gaming session

● Sound: Upon finishing a game session, a 400Hz tone is reproduced. The samples of a the sinusoidal wave stored in memory are transmitted to the DAC thanks to a DMA channel

● Start and reset button

List of components:

- LPC1769 (revB version)
- 8x8 LED matrix, model: KYX-1088AB
- 8 resistors 430 Ω
- 10KΩ potentiometer
- 5 buttons
- earphone for audio output
- UART TTL to USB module type A PL2303

Game logic and mechanics

The game code defines at first, a series of structures and global variables, which are used by functions that model the game's development. The description of each of these variables and structures is provided through comments:

```
#define ANCHO 8
#define ALTO 8
```

```
//Ubicaciones dentro de la matriz 8x8
typedef struct {
    uint8_t x, y;
} Point;

//Direcciones en la que puede moverse la vibora
typedef enum {
    ARRIBA, ABAJO, IZQ, DER
} Direction;

typedef enum {
    EASY, NORMAL, HARD
} Difficulty;

Point snake[ANCHO * ALTO];  // Arreglo de posiciones ocupadas por la vibora
uint8_t snakeLength;        // Cantidad de posiciones ocupadas por la vibora
Point apple;                // Ubicación actual de la manzana a comer
Direction direction;        // Direccion actual en la que se mueve la vibora
uint8_t appleCounter = 0;   // Cantidad de manzanas ya comidas
uint16_t secondsCounter;    // Duración de la partida en segundos
Difficulty difficulty;      // Dificultad actual de la partida
Bool start = FALSE;         // Bandera de comienzo de la partida inicial
```

Image 1: Global structures and variables.

main():

First, it shifts the samples of the sinusoidal signal in 6 positions using a for loop. The function of these samples and why they are shifted is thoroughly explained in the section "Audio Playback via DMA and DAC." Next, the configuration functions for UART, motion buttons, GPIO pins (which control the LED matrix) and ADC are called, along with *helloWorld()* function that displays game instructions upon starting the game for the first time. Finally, it waits until the start flag is set to 1 before entering a while loop, where the render() function is called after a *delay(100)* which lasts approximately 1.2 milliseconds.

```c
int main() {
    // se desplazan las muestras en 6 posiciones
    // VALUE en el registro DACR comprende los bits 15-6
    for(uint8_t index = 0; index<SAMPLES_AMOUNT; index++){
        sinSamples[index] = sinSamples[index]<<6;
    }

    configUART();
    helloWorld();
    configButtons();
    configGPIO();
    configADC();

    while(!start){  //Espero que el boton de start levante la flag antes de continuar con el juego
        delay(300);
    }

    while (1) {     //Una vez configurado e iniciado el juego, renderizo las posiciones de la vibora y la manzana
        render();
        delay(100); //SACAR CUENTAS
    }

    return 0;
}
```

Image 2: main() function.

initGame():

Sets the initial parameters of the snake, calls Timer 0 and SysTick() configuration functions, forces the ADC to perform an single conversion immediately (to set the game speed later), and finally, enables Timer 0.

```c
//Inicializa todo lo necesario para una nueva partida

void initGame(){
    // Setea la longitud de la vibora y su dirección
    snakeLength = 3;
    direction = DER;
    appleCounter = 0;   //Resetea los contadores de manzanas comidas y segundos transcurridos
    secondsCounter =0;

    snake[0].x = 2; snake[0].y = 4;
    snake[1].x = 1; snake[1].y = 4;
    snake[2].x = 0; snake[2].y = 4;

    apple.x=6;
    apple.y=4;

    configTimer0();                          //Timer encargado del tick de movimiento de la vibora
    ADC_StartCmd(LPC_ADC,ADC_START_NOW);     //Hace una unica conversión para obtener la velocidad de juego
    NVIC_EnableIRQ(ADC_IRQn);
    configSysTick();
    TIM_Cmd(LPC_TIM0,ENABLE);                //Habilita el timer encargado del tick de movimiento de la vibora
}
```

Image 3: initGame() function.

moveSnake()

It is responsible for the snake's motion, updating the position of its head based on the current direction every time a Timer 0 interrupt occurs. If there's no collision (*checkCollisions()==false*) the positions are shifted in the corresponding direction, otherwise, the stopGame() function is called.

```c
// Genera la nueva posición de la vibora y si es válida la actualiza en el arreglo snake
void moveSnake(){
    Point newPos = snake[0]; //Copia de la posición actual de la cabeza de la vibora
    if(direction==ARRIBA){
        newPos.y++;
    } else if(direction==ABAJO){
        newPos.y--;
    } else if(direction==DER){
        newPos.x++;
    } else{     //direction==IZQ
        newPos.x--;
    }

    if(!checkCollisions(newPos)){ //Chequeo si estoy haciendo un movimiento válido
        for(int i=snakeLength-1;i>0;i--){ //Recorro el arreglo en orden inverso
            snake[i]=snake[i-1];    //Muevo las posiciones de la vibora un lugar dentro del array
        }
        snake[0]=newPos; //Guardo la nueva posición de la cabeza de la vibora
    } else{
        stopGame();
    }
}
```

Image 4: moveSnake() function.

```c
void TIMER0_IRQHandler(){
    moveSnake();
    TIM_ClearIntPending(LPC_TIM0,TIM_MR0_INT);
}
```

Image 5: timer 0 handler moveSnake().

checkCollisions():

1. **Collision with boundaries:** If the new position is outside the matrix boundaries, it returns -1.

2. **Self-collision:** If the new position coincides with any of the snake's body positions, it returns -1.

3. Snake eats an apple: If the new position coincides with that of the apple, the length of the snake is updated, the counter of eaten apples is incremented, a new apple is generated, and returns 0.

4. Movement to an empty space: If none of the above situations occur, the function returns 0, indicating that there is a valid movement to an empty space.

```
/*  Chequea si el proximo movimiento newPos de la vibora contra cuatro situaciones:
    1) Fuera de los limites -> GameOver: Sonido + StopTotal + Enviar stats
    2) Choque contra si misma -> GameOver
    3) Choque con la manzana -> moveSnake + createNewApple + updateLength
    4) Espacio libre -> moveSnake
    ---
       Return: -1 = Game Over // 0 = moveSnake
*/
uint8_t checkCollisions(Point newPos){
    //Caso 1: Fuera de los limites
    if(newPos.x>=ANCHO || newPos.x<0 || newPos.y>=ALTO || newPos.y<0){
        return -1;
    }
    //Caso 2: Choque consigo misma
    for (int i = 0; i < snakeLength; i++){
        if(newPos.x==snake[i].x && newPos.y==snake[i].y){
            return -1;
        }
    }
    //Caso 3: Choque con una manzana
    if (newPos.x==apple.x && newPos.y==apple.y){
        snakeLength++;
        appleCounter++;
        createNewApple();
        return 0;
    }
    //Caso 4: Movimiento válido a espacio vacío
    return 0;
}
```

Image 6: checkCollisions() function.

updateDirection():

Updates the snake's direction if the new value is not equal to the current direction and is also not equal to the direction that should be avoided (for example, if the snake is moving to the right, it cannot move to the left and vice versa).

```c
/*  Cuando el pulsador envia su dirección correspondiente, chequea si es valido
    y si es así actualiza la dirección actual de la vibora
*/
void updateDirection(Direction new, Direction avoid){
    if(direction!= new && direction!=avoid){
        direction=new;
    }
}
```

Image 7: updateDirection() direction.

createNewApple():

Generates a new apple by calling getRandomPair(), ensuring that its position does not coincide with any of the positions already occupied by the snake.

```c
/*  Genera posición random para nueva manzana
    - Chequea si la posición está ocupada por la vibora
    - Update de la posición al elemento "apple"
*/
void createNewApple(){
    Point newApple;
    uint8_t flag = 1;
    while(flag!=0){
        flag = 0;
        getRandomPair(&newApple.x,&newApple.y);
        for(int i=0;i<snakeLength;i++){ //Verifico la posición de newApple contra todas las de la vibora
            if(snake[i].x==newApple.x && snake[i].y==newApple.y){
                flag++; //Si encuentra una coincidencia, levanto la bandera
            }
        }
    }
    apple=newApple; //Guardo la nueva posición
}
```

Imagen 8: createNewApple() function.

getRandomPair()

Generates a pair of random values between 0-7 (corresponding to a position within the LED matrix), using the current value of SysTick as the seed, which interrupts every 1 millisecond.

```
//Usa el value de Systick para generar dos valores en el rango [0:7] que guarda en a y b
void getRandomPair(uint8_t* a, uint8_t* b){
    volatile uint32_t seed = SysTick->VAL;
    seed ^= (seed << 13);

    *a = (seed & 0x07);
    *b = ((seed >> 3) & 0x07);
}
```

Imagen 9: getRandomPair() function.

helloWorld() y sendStats():

These functions use the UART module to send information to the PC. helloWorld() sends the initial greeting and game instructions when the circuit is powered on for the first time (no image is attached due to its horizontal length, but it can be viewed in detail starting from line 421). On the other hand, sendStats() is responsible for sending game statistics at the end of the game:

- Game ID
- Selected difficulty
- Duration in seconds, counted using SysTick(), which interrupts every 1 millisecond
- Number of apples eaten

To display integers in the console, the uint16_to_uintArray function is used, which converts an integer into an array.

```
//Lleva la cuenta de los segundos de la partida
void SysTick_Handler(){
    static uint16_t millisCount = 0;
    millisCount++;

    if(millisCount >= 1000){
        secondsCounter++;
        millisCount = 0;
    }
}
```

Image 10: SysTick handler and miliseconds counter

```c
//Se encarga de enviar las estadisticas de la partida a la PC
void sendStats(){
    static uint8_t  gameCounter = 0;    //Contador de partidas
    gameCounter++;
    uint8_t numbers[4]; //Buffer para el array de digitos
    uint8_t digitos;    //Auxiliar para contar la cantidad de digitos en el buffer

    uint8_t data0[] = "\n\rChan chan chan...Se terminó el juego mi loco! Acá van un par de estadisticas:\n\r";
    UART_Send(LPC_UART1,data0, sizeof(data0), BLOCKING);

    UART_Send(LPC_UART1,(uint8_t*)" ID de partida: ",17, BLOCKING);
    digitos=uint16_to_uint8Array(gameCounter, numbers);
    UART_Send(LPC_UART1,(uint8_t *)numbers, digitos, BLOCKING);

    UART_Send(LPC_UART1,(uint8_t*)"\n\r Dificultad seleccionada: ",29, BLOCKING);
    if(difficulty==EASY){
        UART_Send(LPC_UART1,(uint8_t *)"FACIL",5,BLOCKING);
    } else if(difficulty==NORMAL){
        UART_Send(LPC_UART1,(uint8_t *)"NORMAL",6,BLOCKING);
    } else{
        UART_Send(LPC_UART1,(uint8_t *)"DIFICIL",7,BLOCKING);
    }
    UART_Send(LPC_UART1,(uint8_t*)"\n\r Duración de la partida en segundos: ",41, BLOCKING);
    digitos=uint16_to_uint8Array(secondsCounter, numbers);
    UART_Send(LPC_UART1,(uint8_t *)numbers, digitos, BLOCKING);

    UART_Send(LPC_UART1,(uint8_t*)"\n\r Manzanas comidas: ",22, BLOCKING);
    digitos=uint16_to_uint8Array(appleCounter, numbers);
    UART_Send(LPC_UART1,(uint8_t *)numbers, digitos, BLOCKING);

    UART_Send(LPC_UART1,(uint8_t *)"\n\r",2,BLOCKING);
}
```

Image 11: sendStats() function.

```c
//Convierte un entero de 16bits a un string de digitos en ASCII
uint8_t uint16_to_uint8Array(uint16_t value, uint8_t *result){
// Buffer size based on the maximum number of digits in a uint16_t (5 digits)
    uint8_t buffer[5];
    // Initialize index
    uint8_t index = 0;

    // Handle the case when the value is 0 separately
    if (value == 0) {
        buffer[index++] = '0';
    } else {
        // Extract digits in reverse order
        while (value > 0) {
            buffer[index++] = '0' + (value % 10);
            value /= 10;
        }
    }
    // Reverse the buffer to get the correct order
    for (int8_t i = 0; i < index; ++i) {
        result[i] = buffer[index - 1 - i];
    }
    // Null-terminate the result
    result[index] = '\0';
    return index;
}
```

Imagen 12: uint16_to_uint8Array() function.

render():

Is responsible for updating the matrix to display the current state of the snake and the apple. The LED matrix is represented by 2 arrays:

● X represents the columns. Each element of this array indicates which pins of port 2 (specified in the “8x8 LED Matrix section”) should be set high, with the rest set to 0.

● Y represents the rows. Each element of this array indicates which pin of port 0 (specified in the “8x8 LED Matrix section”) should be set low, leaving the rest set to 1.

When these arrays are traversed from left to right, the movement along the matrix is as shown in the following image.

```
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};
```

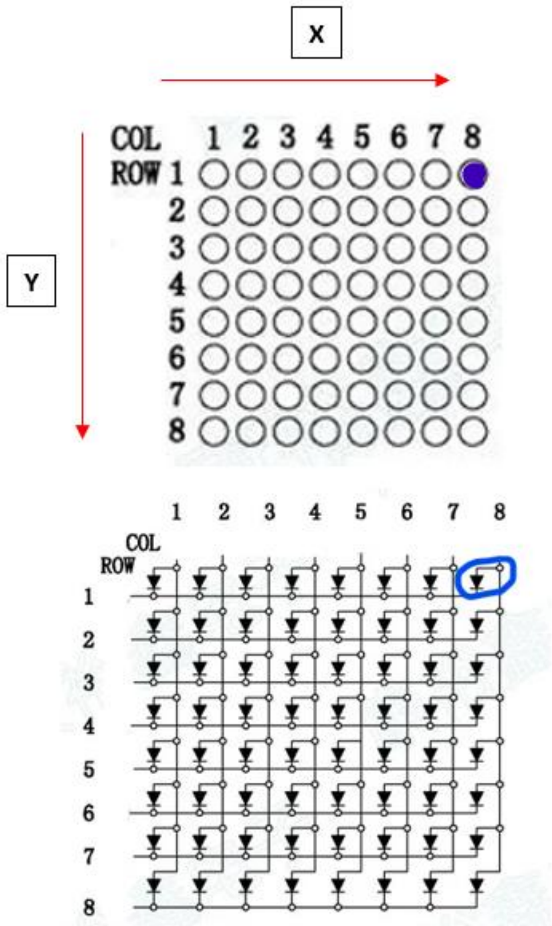


Image 13: X and Y positions array.

Therefore, if you want to, for example, turn on the LED located in column 8 and row 1, set port 2 to 0x0080 and port 0 to 0x0DF0.

Using a static variable, a particular LED corresponding to a snake position or the apple position is illuminated with each call to this function. **The calls are made at a speed high enough for the human eye to perceive that multiple LEDs are lit simultaneously** (see the delay() function in main).

```
//Chequea y envía los leds a encender a la matriz
void render(){

    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};

    static int i = 0;
    if(i>=(snakeLength+1)){
        i=0;
    }

    if(!i){      //Renderizar posición de la manzana
        LPC_GPIO2->FIOPINL = X[apple.x];
        LPC_GPIO0->FIOPINL = Y[apple.y];
    } else{
        LPC_GPIO2->FIOPINL = X[snake[i-1].x];
        LPC_GPIO0->FIOPINL = Y[snake[i-1].y];
    }
    i++;
}
```

Image 14: Función render() function.

stopGame():

Disables Timer 0 and SysTick to stop the movement of the snake and the counting of seconds, respectively. The game statistics are sent using sendStats(), and finally, the DAC and DMA channel configuration functions are called (to reproduce the end-of-game sound).

```
void stopGame(){
    TIM_Cmd(LPC_TIM0,DISABLE);
    SYSTICK_Cmd(DISABLE);
    sendStats();
    configDAC();
    configDMA_DAC_Channel();
}
```

Image 15: stopGame() function

8x8 Led matrix and pins that control it

The game is displayed on a 8x8 matrix of blue LEDs (common anode). The pin diagram is shown below:

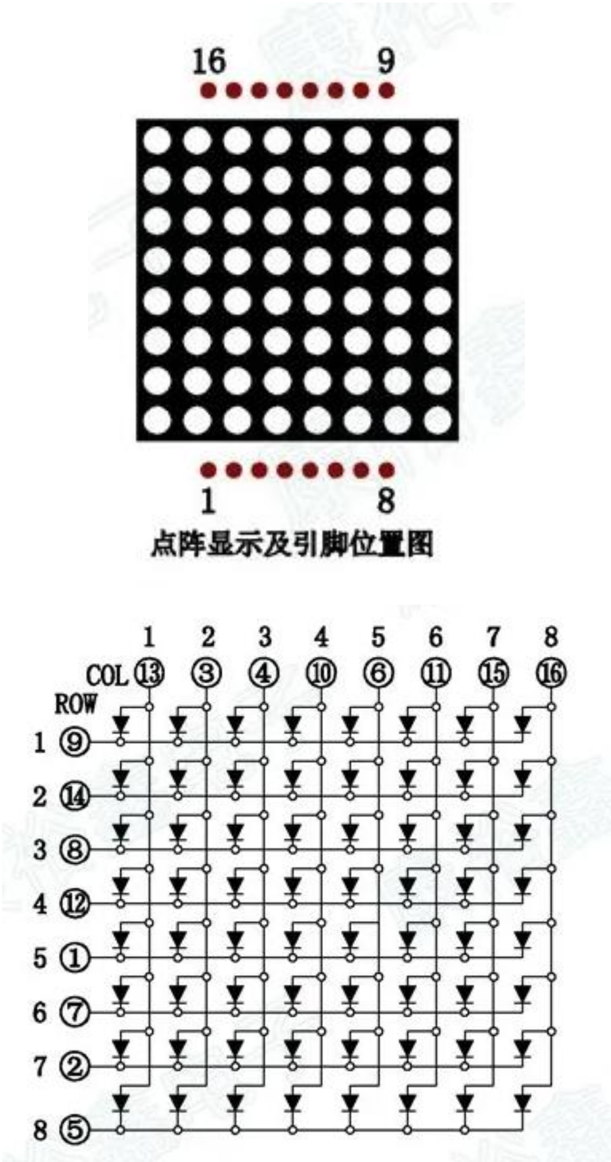


Image 16: LED Matrix,model KYX-1088AB

P2.0 to P2.7 are used used to control which pins should be set high (refer to image 16, those that "select" a particular column). P0.4 to P0.11 on the other hand, determine which pins of the matrix should be set low to access a specific row.

Pin of the LED matrix that is set high	16	15	13	11	10	6	4	3
Pin of the board that controls it	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0

Pin of the led matrix that is set low	14	12	9	8	7	5	2	1
Pin of the board that controls it	P0.11	P0.10	P0.9	P0.8	P0.7	P0.6	P0.5	P0.4

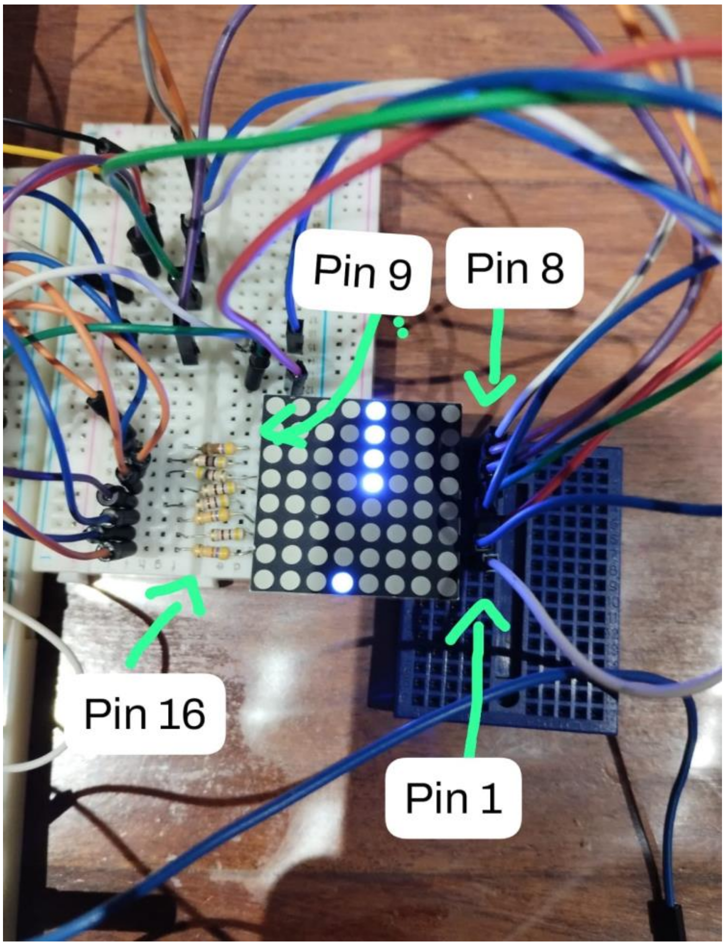


Image 17: LED matrix, 430-ohm resistors are placed to limit the current reaching the LEDs.

Motion buttons and Start/Restart button (GPIO)

One way to control the snake's motion is through push buttons. Pins P0.0-P0.7 are configured to generate interrupts (GPIO) when a rising edge occurs on any of them. The *EINT3_IRQHandler()* changes the snake's direction as appropriate.

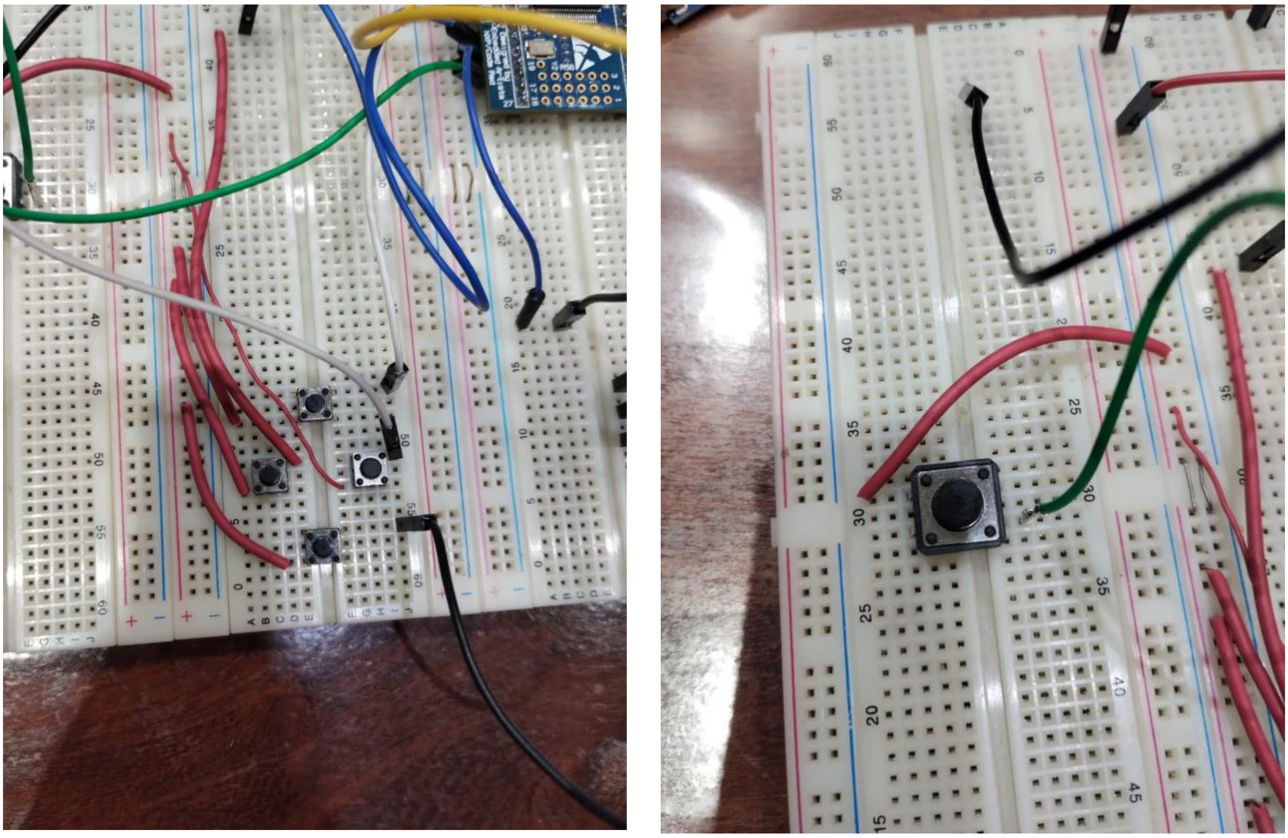
The start/restart button is implemented using P0.22, and just like the motion controls, rising edge interrupts are enabled. Within the button's interrupt routine, the *start* flag is set high if it is not already (see function *main()*) and the initGame() function is called to start the game. Both the movement buttons and the restart button implement integrated pull down resistors in order to avoid bouncing issues.

```
void configButtons(){
    //Se habilita resistencias de pull down en los pines P0.0 a P0.3
    // P0.0----->ARRIBA
    // P0.1----->DERECHA
    // P0.2----->IZQUIERDA
    // P0.3----->ABAJO
    LPC_PINCON->PINMODE0|=(3<<0);
    LPC_PINCON->PINMODE0|=(3<<2);
    LPC_PINCON->PINMODE0|=(3<<4);
    LPC_PINCON->PINMODE0|=(3<<6);

    // Se habilita resistencia de pull down en P0.22
    // P0.22----->START/RESTART
    LPC_PINCON->PINMODE1|=(3<<12);

    // Se habilita interrupción por flanco de subida en todos los botones
    LPC_GPIOINT->IO0IntEnR |=0x0000000F;
    LPC_GPIOINT->IO0IntEnR |=(1<<22);

    // Se limpian banderas de interrupción
    LPC_GPIOINT->IO0IntClr |=0x0000000F;
    LPC_GPIOINT->IO0IntClr |=(1<<22);

    NVIC_EnableIRQ(EINT3_IRQn);
}
```

Image 18: Motion and start/reset buttons configuration function

```c
void EINT3_IRQHandler(){
    //ARRIBA
    if((LPC_GPIOINT->IO0IntStatR)&(1<<0)){
        updateDirection(DER,IZQ);
        LPC_GPIOINT->IO0IntClr |=(1<<0);
    }
    //DERECHA
    else if((LPC_GPIOINT->IO0IntStatR)&(1<<1)){
        updateDirection(ARRIBA,ABAJO);
        LPC_GPIOINT->IO0IntClr |=(1<<1);
    }
    //IZQUIERDA
    else if((LPC_GPIOINT->IO0IntStatR)&(1<<2)){
        updateDirection(ABAJO,ARRIBA);
        LPC_GPIOINT->IO0IntClr |=(1<<2);
    }
    //ABAJO
    else if((LPC_GPIOINT->IO0IntStatR)&(1<<3)){
        updateDirection(IZQ,DER);
        LPC_GPIOINT->IO0IntClr |=(1<<3);
    }
    // BOTON DE START/RESTART
    else{
        if(!start){
            start = TRUE;
        }
        initGame();
        LPC_GPIOINT->IO0IntClr |=(1<<22);
    }
}
```

Image 19: LED matrix, model KYX-1088AB



Images 20 y 21: movement buttons and start/restart button

Speed regulator (ADC and Potentiometer)

The ADC channel 0 is configured to perform a single conversion and trigger an interrupt upon completion (see the initGame() function in the "Game Logic and Mechanics" section). The acquired value is used to set the corresponding timer 0 match value before enabling it and, therefore, determine the game's speed. If the sampled value is greater than 3500, the selected difficulty is hard; if it is greater than 1000 and less than 3500, it is normal; and if it is below 1000, the game enters easy mode.

```c
void configADC(){
    PINSEL_CFG_Type PinCfg;
    PinCfg.Funcnum = 1;
    PinCfg.OpenDrain = 0;
    PinCfg.Pinmode = 0;          //Sin pull-up ni pull-down
    PinCfg.Pinnum = 23;
    PinCfg.Portnum = 0;
    PINSEL_ConfigPin(&PinCfg);   //P0.23 como AD0.0

    ADC_Init(LPC_ADC, 200000);                          //Frec. de muestreo = 200kHz
    ADC_IntConfig(LPC_ADC,ADC_ADINTEN0,ENABLE);         //Habilito interrupción canal 0
    ADC_ChannelCmd(LPC_ADC,ADC_CHANNEL_0,ENABLE);       //Habilito canal 0
}
```

Imagen 22: ADC 0 Channel configuration

```c
#define HARD_MAX     3500
#define NORMAL_MAX   1000

#define TIMER_EASY    1300
#define TIMER_NORMAL  800
#define TIMER_HARD    400
```

Imagen 23: Constants

```c
void ADC_IRQHandler(){
    __IO uint32_t adcValue = 0; //Uso variable local, no hay necesidad de tenerla como global
    if (ADC_ChannelGetStatus(LPC_ADC,ADC_CHANNEL_0,ADC_DATA_DONE)){ //Leo el valor de connversión en el canal 0
        adcValue =  ADC_ChannelGetData(LPC_ADC,ADC_CHANNEL_0);
        NVIC_DisableIRQ(ADC_IRQn);
    }

    //A menor valor en la medición, mayor es la resistencia del potenciometro
    //Escala de mediociones del ADC: 0--(Zona dificil)--HARD_MAX--(Zona normal)--NORMAL_MAX--(Zona facil)--4095

    if(adcValue>HARD_MAX){          //El potenciometro está cerca de su valor minimo
        TIM_UpdateMatchValue(LPC_TIM0,0,TIMER_HARD);
        difficulty = HARD;
    } else if(adcValue>NORMAL_MAX){   //El potenciometro está en un valor intermedio
        TIM_UpdateMatchValue(LPC_TIM0,0,TIMER_NORMAL);
        difficulty = NORMAL;
    } else{                           //El potenciometro está cerca de su valor maximo
        TIM_UpdateMatchValue(LPC_TIM0,0,TIMER_EASY);
        difficulty = EASY;
    }

    LPC_ADC->ADGDR &= LPC_ADC->ADGDR;
}
```

Image 24: Difficulty selection and timer 0 match value configuration

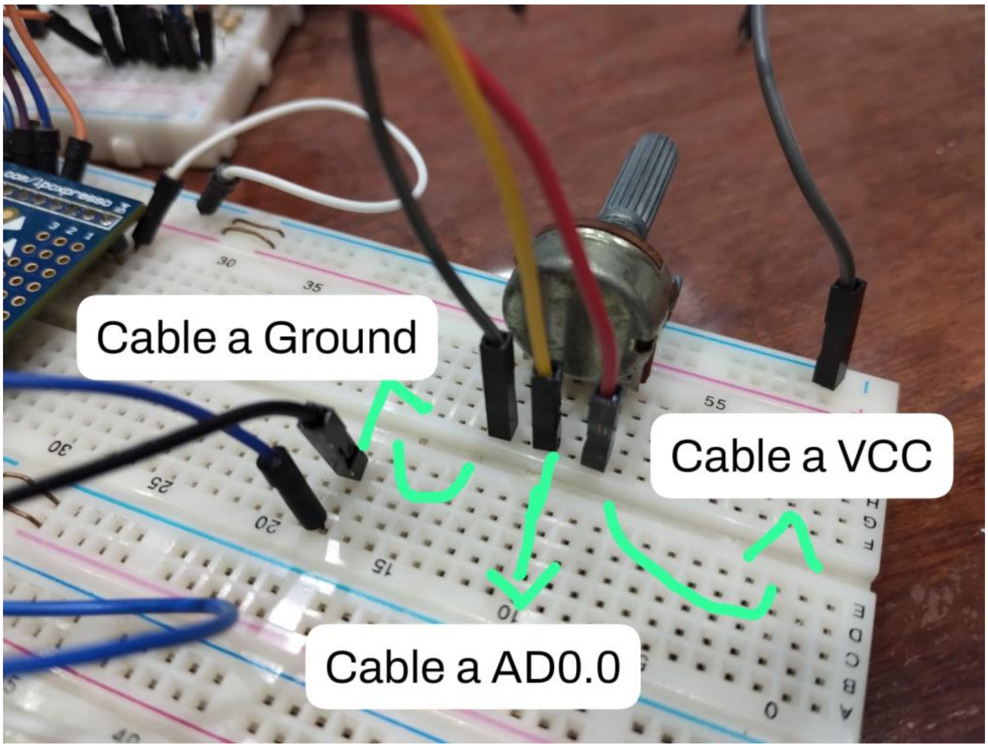


Image 25: Potentiometer Connections

Sound playback via DAC and DMA

At the end of the game (as it could be seen in the *stopGame()* function), a sinusoidal wave is played through the DAC by transferring the samples from the array *sinSamples[ ]* to this peripheral through DMA channel 0. Each sample of the sinusoidal signal is shifted by 6 positions in the main() function, this is because the 10-bit value to be loaded into the DAC (to produce an analog output), is within bits [15-6] of the DACR register. The sound is played through a earphone and only stops when a new game is initiated.

Calculations:

Given that the emitted wave has a frequency of 400Hz and there are a total of 60 samples, the timeout time of the DAC (the time elapsed between DMA data transfer requests) is calculated as follows:

$$60\ samples - - - - - - - - \frac{1}{400Hz}$$

$$1\ sample - - - - - - - - X$$

$$X = \frac{\frac{1}{400Hz}}{60} = 4{,}166 * 10^{-5}s = time\ between\ samples$$

To generate this time value, it should be noted that the CPU clock is 100MHz, and PCLK_DAC is equal to the mentioned value divided by 4. Therefore:

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

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

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

This is the value that, when loaded into the time out counter, will generate a DMA transfer request every $4{,}166 * 10^{-5}s$.

Configuration:

The complete configuration of the DAC and DMA channel is carried out using the functions configDAC() and configDMA_DAC_Channel()."

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

Image 26: Constants for DAC time-out counter value calculation.

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

Image 27: Sinusoidal wave samples

```
void configDAC(){
    //Configuración de P0.26 como salida analógica del DAC
    PINSEL_CFG_Type pinCfg;
    pinCfg.Funcnum = 2;
    pinCfg.OpenDrain = 0;
    pinCfg.Pinmode = 0;
    pinCfg.Portnum = 0;
    pinCfg.Pinnum = 26;
    PINSEL_ConfigPin(&pinCfg);

    DAC_CONVERTER_CFG_Type dacCfg;
    dacCfg.CNT_ENA = SET;
    dacCfg.DMA_ENA = SET;
    DAC_Init(LPC_DAC);

    // configuración tiempo de time out DAC
    uint32_t tmp;
    tmp = (PCLK_DAC_IN_MHZ * 1000000)/(SINE_FREQ_IN_HZ * SAMPLES_AMOUNT);
    DAC_SetDMATimeOut(LPC_DAC, tmp);
    DAC_ConfigDAConverterControl(LPC_DAC, &dacCfg);
}
```

Image 28: DAC output pin and parameters configuration .

```c
void configDMA_DAC_Channel(){
    /*---------Configuración linked list--------*/
    //source width 32 bits
    //destination width 32 bits
    //source adress se incrementa en cada transmisión
    //destination adress (DAC) se mantiene fija
    GPDMA_LLI_Type LLI1;
    LLI1.SrcAddr = (uint32_t) sinSamples;
    LLI1.DstAddr = (uint32_t) &LPC_DAC->DACR;
    LLI1.NextLLI = (uint32_t) &LLI1;
    LLI1.Control = SAMPLES_AMOUNT| (1<<19)  | (1<<22) | (1<<26);

    GPDMA_Init();

    // configuracion y habilitacion del Canal 0 de DMA
    GPDMA_Channel_CFG_Type GPDMACfg;
    GPDMACfg.ChannelNum = 0;
    GPDMACfg.SrcMemAddr = (uint32_t)sinSamples;
    GPDMACfg.DstMemAddr = 0;
    GPDMACfg.TransferSize = SAMPLES_AMOUNT;
    GPDMACfg.TransferWidth = 0;
    GPDMACfg.TransferType = GPDMA_TRANSFERTYPE_M2P;
    GPDMACfg.SrcConn = 0;
    GPDMACfg.DstConn = GPDMA_CONN_DAC;
    GPDMACfg.DMALLI = (uint32_t)&LLI1;
    GPDMA_Setup(&GPDMACfg);
    GPDMA_ChannelCmd(0, ENABLE);
}
```

Image 29: DMA Channel 0 configuration

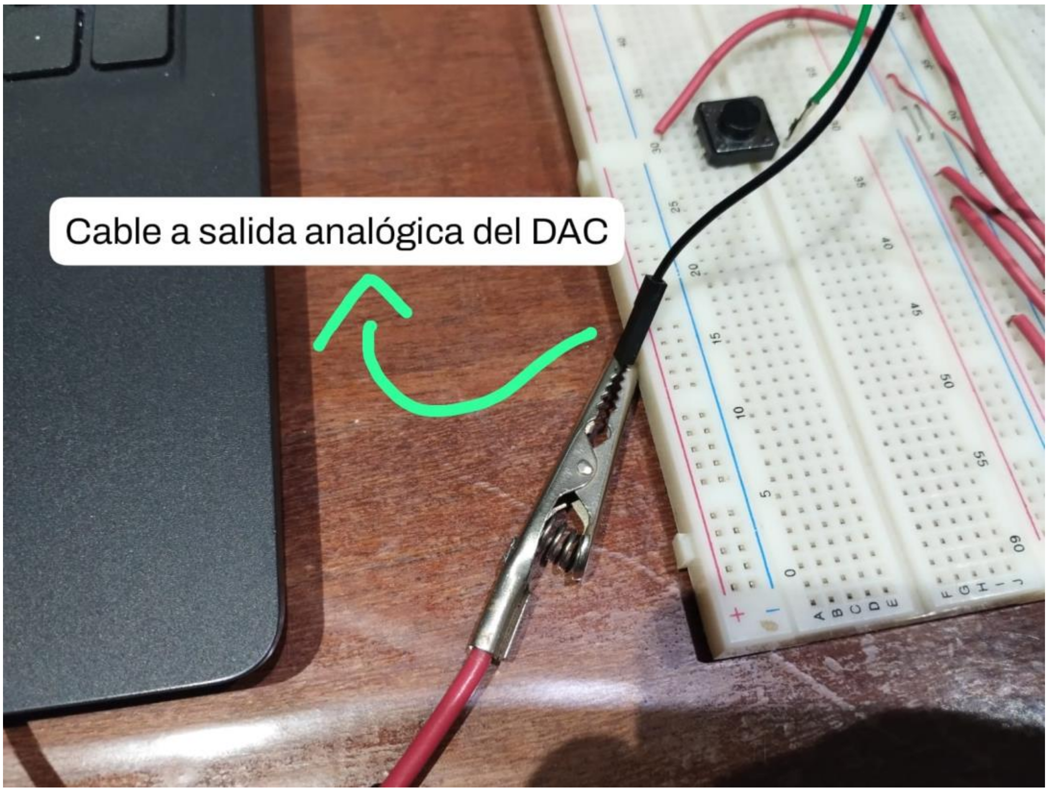


Imagen 30 : DAC output

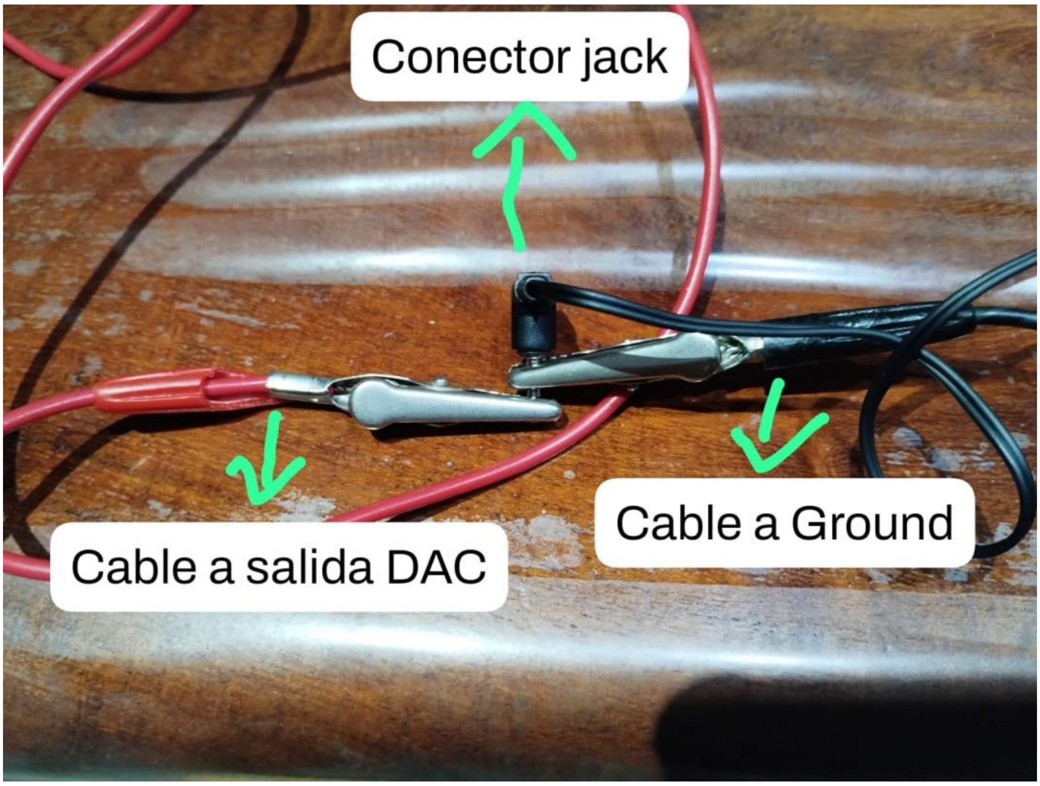


Image 31: Connections to DAC output and ground on earphone jack connector

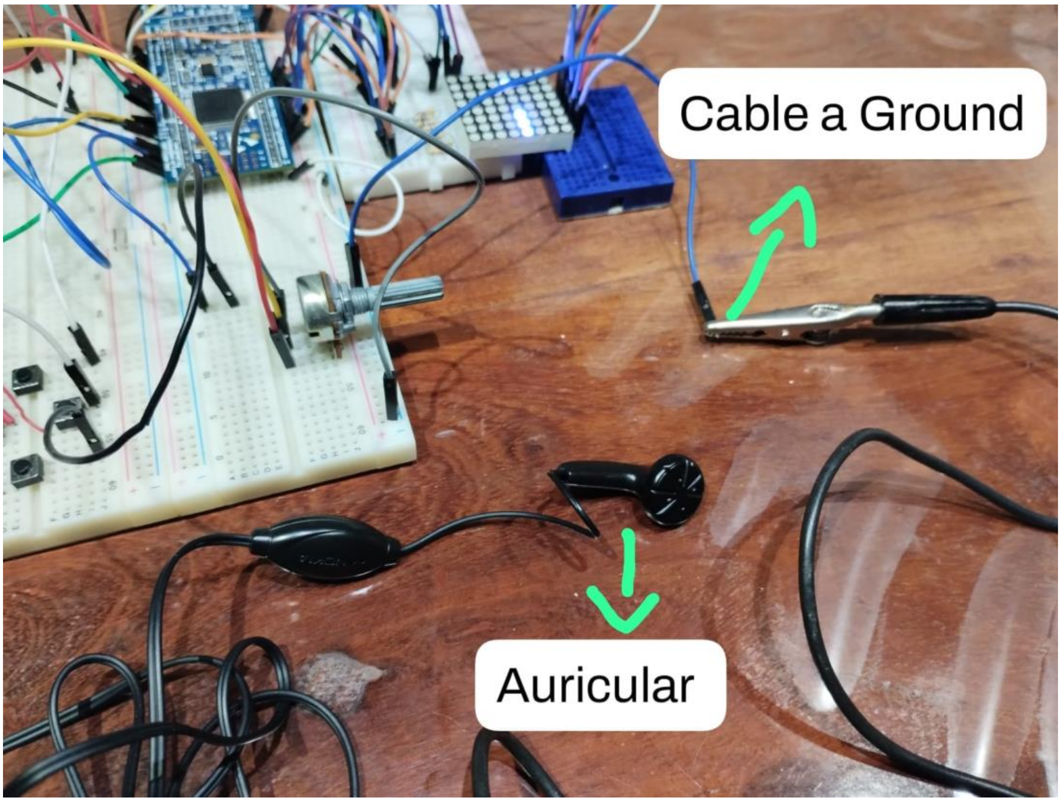


Image 32: Earphone and ground connection

Statistics and Motion Control via UART 1

With the help of a UART to USB type A converter, UART1 module is used, on the one hand, to transmit statistics and game instructions to the PC through the functions helloWorld() and sendStats(). These functions were detailed in the **'Game Logic and Mechanics'** section.

The second task of this peripheral is to receive data from the computer, specifically the characters 'WASD', allowing the player to choose between 2 forms of snake movement control. Each time a character is entered through the console, an interruption is generated, and in its handler, the received ASCII character is identified by calling the *ASCCItoDirection()* function. If it is not valid, the snake's course remains unaffected. The UART1 is configured with the *configUART()* function.

```
void configUART(){
    //Configuro los pines Rx y Tx
    PINSEL_CFG_Type PinCfg;
    PinCfg.Funcnum = 1;
    PinCfg.OpenDrain = 0;
    PinCfg.Pinmode = 0;
    PinCfg.Pinnum = 15;
    PinCfg.Portnum = 0;
    PINSEL_ConfigPin(&PinCfg); //P0.15
    PinCfg.Pinnum = 16;
    PINSEL_ConfigPin(&PinCfg); //P0.16

    UART_CFG_Type UARTConfigStruct;
    UART_FIFO_CFG_Type UARTFIFOConfigStruct;
    UART_ConfigStructInit(&UARTConfigStruct);    //Usamos la configuración por defecto
    UART_Init(LPC_UART1, &UARTConfigStruct);

    UART_FIFOConfigStructInit(&UARTFIFOConfigStruct);
    UART_FIFOConfig(LPC_UART1, &UARTFIFOConfigStruct);

    // Habilita interrupci�n por el RX del UART
    UART_IntConfig(LPC_UART1, UART_INTCFG_RBR, ENABLE);
    // Habilita interrupci�n por el estado de la linea UART
    UART_IntConfig(LPC_UART1, UART_INTCFG_RLS, ENABLE);

    UART_TxCmd(LPC_UART1, ENABLE);  //Habilitamos la transmisión
    NVIC_EnableIRQ(UART1_IRQn);
}
```

Image 33: UART1 module configuration

```c
void UART1_IRQHandler(void){
    uint8_t data[1] = "";

    UART_Receive(LPC_UART1, data, sizeof(data), NONE_BLOCKING);
    ASCIItoDirection(data[0]);

}
```

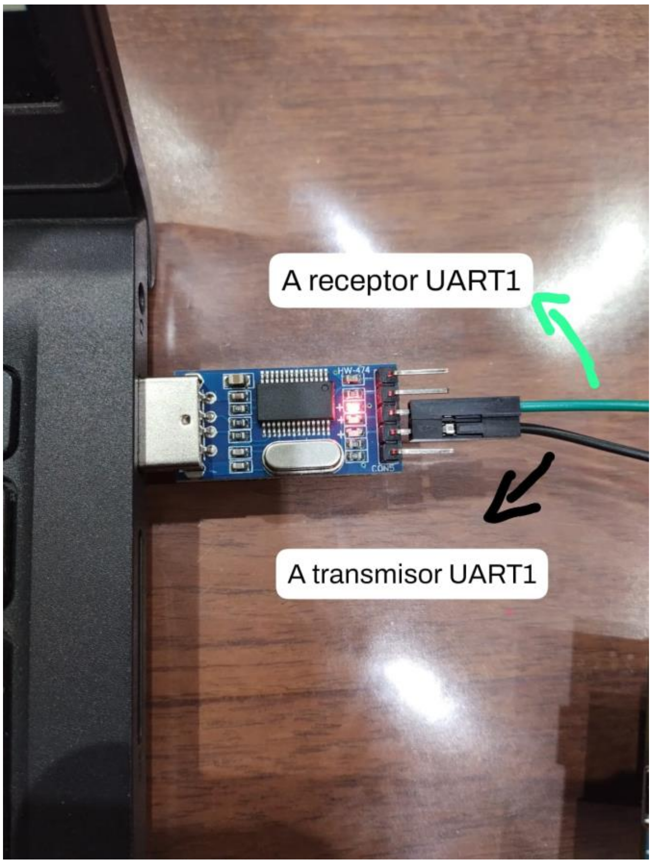
Image 34: UART1 handler

```c
void ASCIItoDirection(uint8_t value){
    if(value=='w'){
        updateDirection(DER,IZQ);
    } else if(value=='s'){
        updateDirection(IZQ,DER);
    } else if(value=='a'){
        updateDirection(ABAJO,ARRIBA);
    } else if(value=='d'){
        updateDirection(ARRIBA,ABAJO);
    }
}
```

Image 35: ***ASCCItoDirection() function***

```
COM3 - PuTTY
Bueeeenas! 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 estadisticas y reproducir un sonido
 - Pero eso no es todo! Queres seguir jugando? Simplemente presioná el boton 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
```

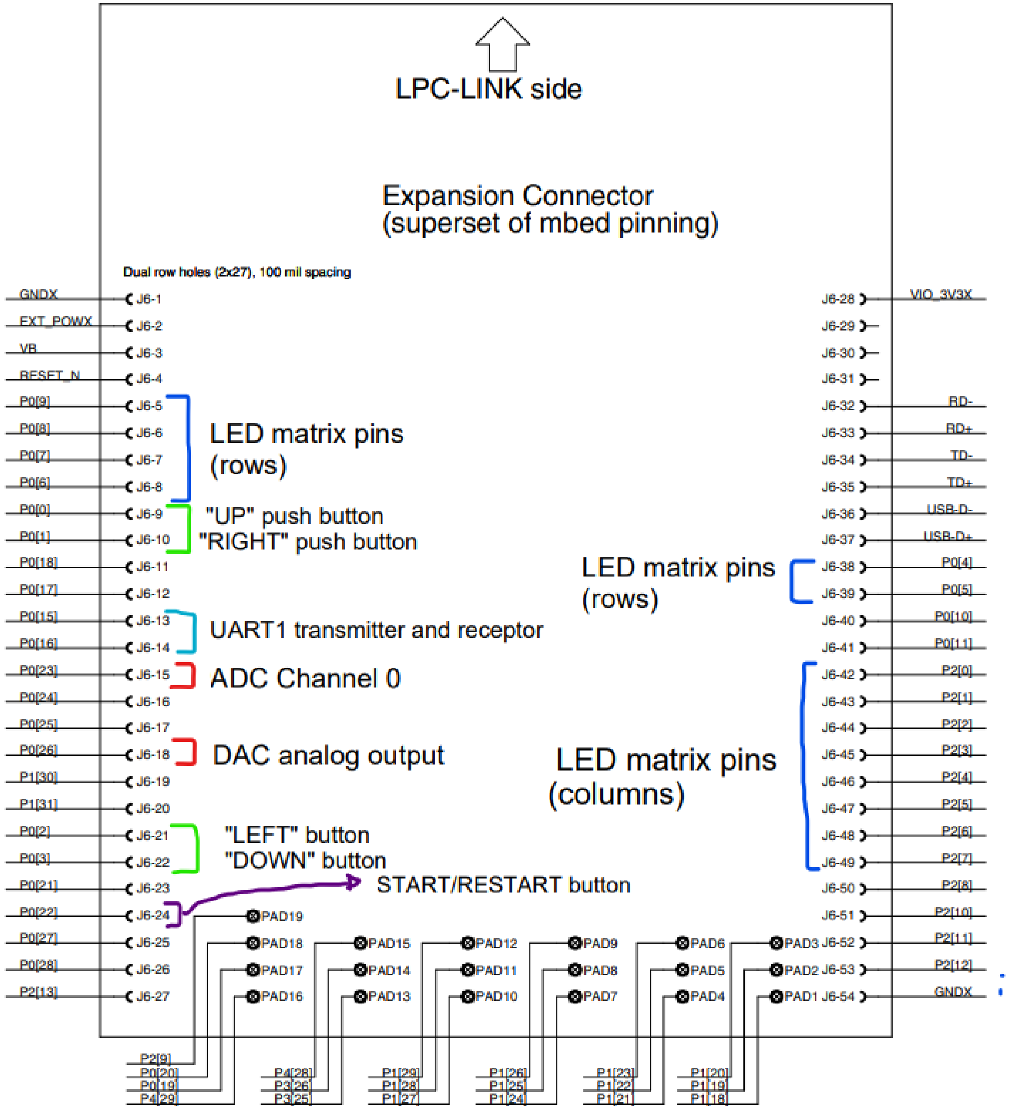
Image 36: Information transmitted from the board to the PC. Includes the game's instructions and the results of 3 matches where the selected difficulty varies.



Images 37 and 38 : UART to USB type A module top and bottom view

Wiring diagram in datasheet

mbed	LPCXpresso	
GND	GND	
VIN (4.5-14V)	VIN (4.5-5.5V)	
VB (battery supply)	VB (battery supply)	
nR (reset)	RESET_N	
SPI1-MOSI	P0.9	MOSI1
SPI1-MISO	P0.8	MISO1
SPI1-SCK	P0.7	SCK1
GPIO	P0.6	SSEL1
UART1-TX / I2C1-SDA	P0.0	TXD3/SDA1
UART1-RX / I2C1-SCL	P0.1	RXD3/SCL1
SPI2-MOSI	P0.18	MOSI0
SPI2-MISO	P0.17	MISO0
SPI2-SCL / UART2-TX	P0.15	TXD1/SCK0
UART2-RX	P0.16	RXD1/SSEL0
AIN0	P0.23	AD0.0
AIN1	P0.24	AD0.1
AIN2	P0.25	AD0.2
AIN3 / AOUT	P0.26	AD0.3/AOUT
AIN4	P1.30	AD0.4
AIN5	P1.31	AD0.5
	P0.2	
	P0.3	
	P0.21	
	P0.22	
	P0.27	
	P0.28	
	P2.13	



LPCXpresso		mbed
VOUT (+3.3V out) if self powered, else +3.3V input		VOUT (3.3V out)
not used		VU (5.0V USB out)
not used		IF+
not used		IF-
RD-		RD- (Ethernet)
RD+		RD+ (Ethernet)
TD-		TD- (Ethernet)
TD+		TD+ (Ethernet)
USB-D-		D- (USB)
USB-D+		D+ (USB)
P0.4	CAN_RX2	CAN-RD
P0.5	CAN_TX2	CAN-TD
P0.10	TXD2/SDA2	UART3-TX / I2C2-SDA
P0.11	RXD2/SCL2	UART3-RX / I2C2-SCL
P2.0	PWM1.1	PWMOUT0
P2.1	PWM1.2	PWMOUT1
P2.2	PWM1.3	PWMOUT2
P2.3	PWM1.4	PWMOUT3
P2.4	PWM1.5	PWMOUT4
P2.5	PWM1.6	PWMOUT5
P2.6		
P2.7		
P2.8		
P2.10		
P2.11		
P2.12		
GND		

Annexes

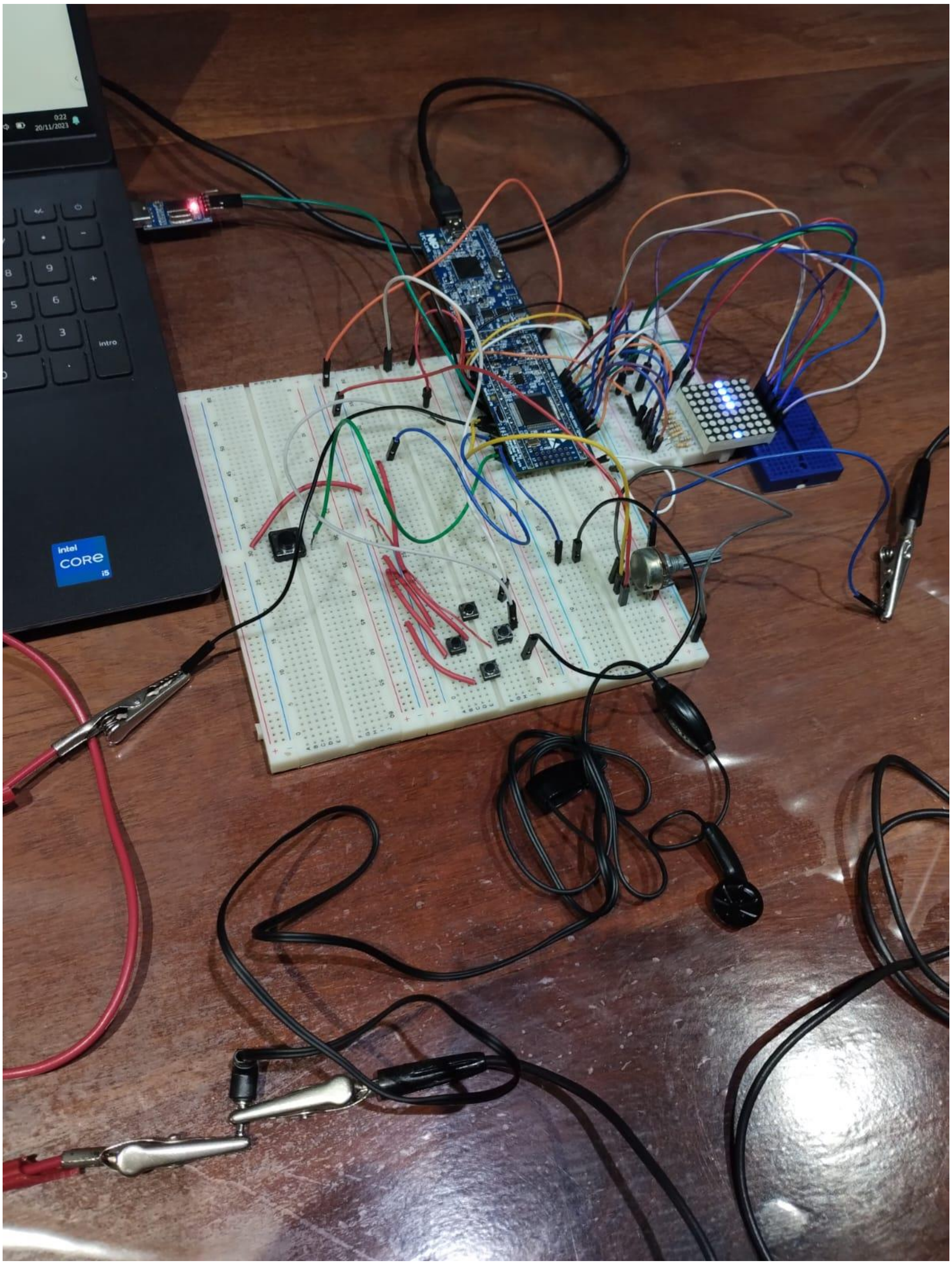


Image 39 : Full circuit

-Led matrix datasheet: https://es.aliexpress.com/item/32594255205.html

-Uart ttl to USB type A module: https://www.todomicro.com.ar/optoacopladores-conversores-y-adaptadores-rs232-rs485-rs422-uart/84-usb-20-a-uart-ttl-5-pines-5v.html