Facultad de
Ciencias Exactas
Físicas y Naturales

Universidad
Nacional
de Córdoba

Informe Trabajo Práctico Final

Sampler de Sonido con LPC1769

Materia: Electrónica Digital III

Estudiantes:
Ramirez, Valentin José
Beierbach, Alejo
Lopez Sivilat, Jose Ignacio

Profesor: Ing. Fernando Gallardo

Resumen.

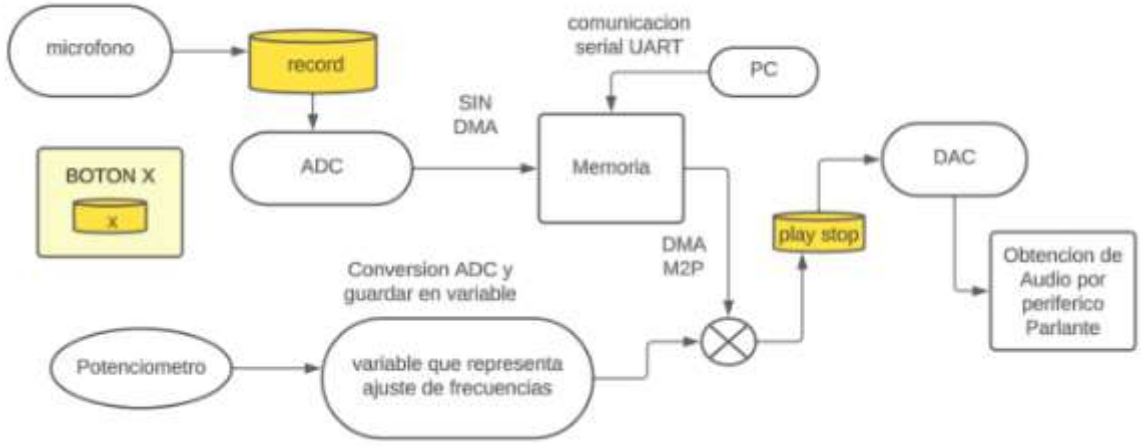
Nuestro proyecto es crear un sampler de sonido que sirva como un medio para implementar conceptos acerca de la placa LPC1769 y otros módulos.

Enunciado

Cuando se pulsa el botón **"Record"**, conectado al pin **P2.10**, se captura la señal detectada por el micrófono, que está conectado al pin **P0.22.** Esta señal analógica se almacena directamente en la memoria, sin la necesidad de utilizar **DMA**, mientras el botón permanece presionado**.** Al iniciar una nueva grabación, la zona de memoria se restablece automáticamente**.** Además, un potenciómetro conectado al pin **P0.24** permite ajustar la frecuencia del sonido mediante la configuración del **timeout** del **DAC.**

Cuando se presiona el botón **"Play/Stop"**, conectado al pin **P2.11**, los datos almacenados en la memoria son extraídos y enviados al **DAC** mediante el uso de **DMA.** La frecuencia de reproducción se determina por el valor ajustado con el potenciómetro en el pin **P0.24**, y la señal de audio resultante se reproduce a través de la salida analógica en el pin **P0.26.** Como método adicional de comunicación, es posible enviar una secuencia de sonidos en forma de valores decimales a través del **UART.** Estos valores se almacenan en la misma sección de memoria utilizada para las grabaciones realizadas con el micrófono**.**

El comportamiento descrito puede representarse en el siguiente diagrama:



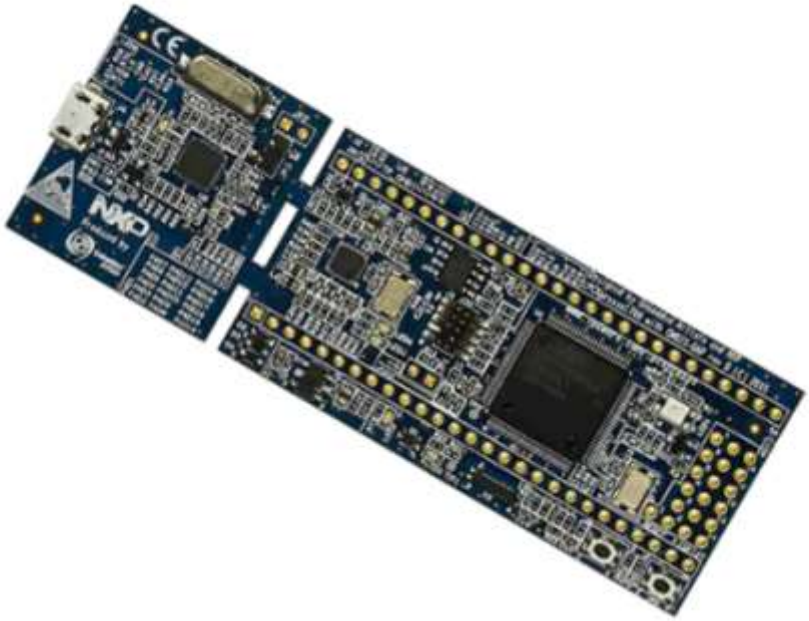
Marco Teórico

Placa LPC1769

El **LPC1769** es un microcontrolador **Cortex®-M3** diseñado para aplicaciones embebidas que requieren un alto nivel de integración y bajo consumo de energía, operando a frecuencias de hasta **120 MHz.** Sus principales características incluyen:

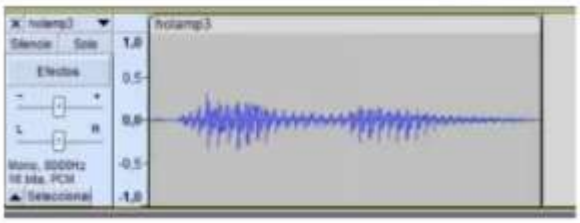
- **Memoria**:
    - **512 kB** de memoria flash.
    - **64 kB** de memoria de datos.
- **Interfaces de comunicación**:
    - **4 UART.**
    - **2 canales CAN.**
    - **3 SSP/SPI**.
    - **3 I2C.**
    - **I2S.**
- **Periféricos avanzados**:
    - **Ethernet MAC.**
    - **Dispositivo/host/OTG USB.**
    - **Controlador DMA** de 8 canales.
- **Convertidores**:
    - **ADC** de 8 canales y 12 bits.
    - **DAC** de 10 bits.
- **Control y temporización**:
    - Control de motor **PWM.**
    - **4 temporizadores** de uso general.
    - Reloj de tiempo real (**RTC**) de potencia ultrabaja con suministro de batería separado.

Esta placa se adapta perfectamente a aplicaciones exigentes en electrónica digital e integración de sistemas.



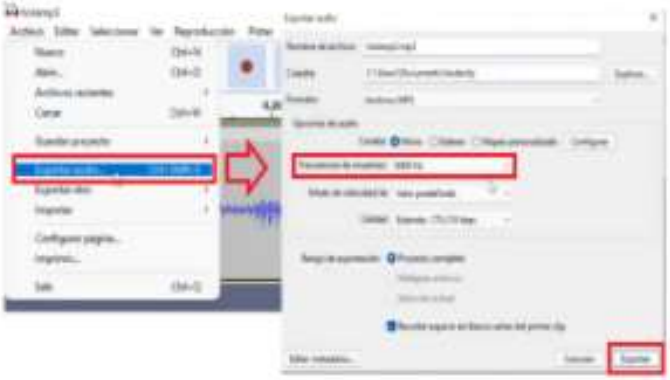
Codificar MP3 a Decimal

A través del software “Audacity” ingresando un fragmento de audio en formato MP3, y con la ayuda del software “Encoder” se puede conseguir la codificación decimal del audio de interés. Los pasos a seguir son los siguientes:



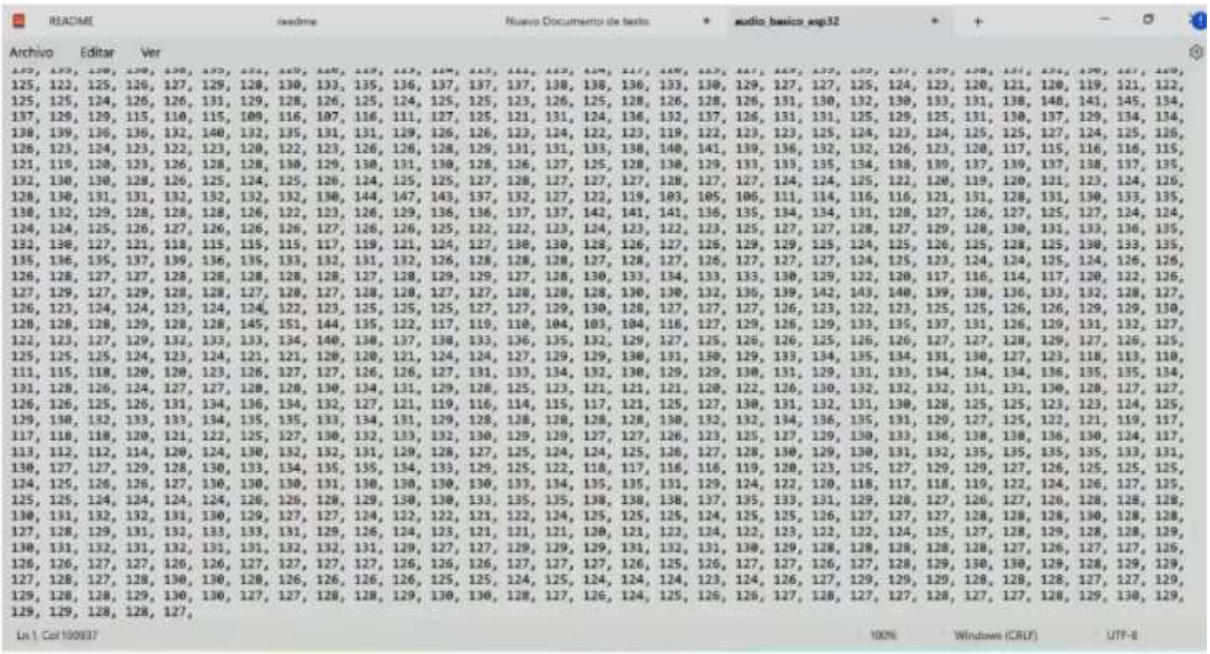
Paso 1: Abrir el archivo de audio “holamp3.mp3” formato MP3 en el software Audacity

Paso 2: Seleccionar el formato de 16 bits PCM



Paso 3: Para exportar el audio, se elige la mínima frecuencia de muestreo posible, esto afecta la calidad de audio, pero permite almacenar mayor cantidad de información decimal en la placa, este es el precio que pagar por usar un sistema con recursos limitados.

Paso 4: Busco el audio (“holamp3.mp3”) con la app Encoder

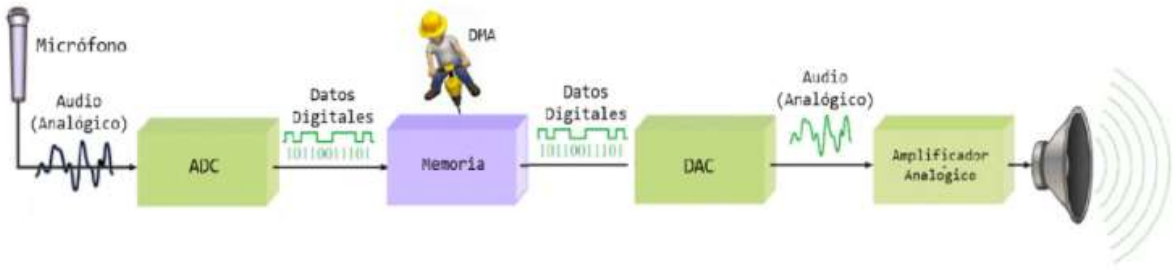


Paso 5: se guarda en el bloc de notas, obteniendo la codificación a decimal.

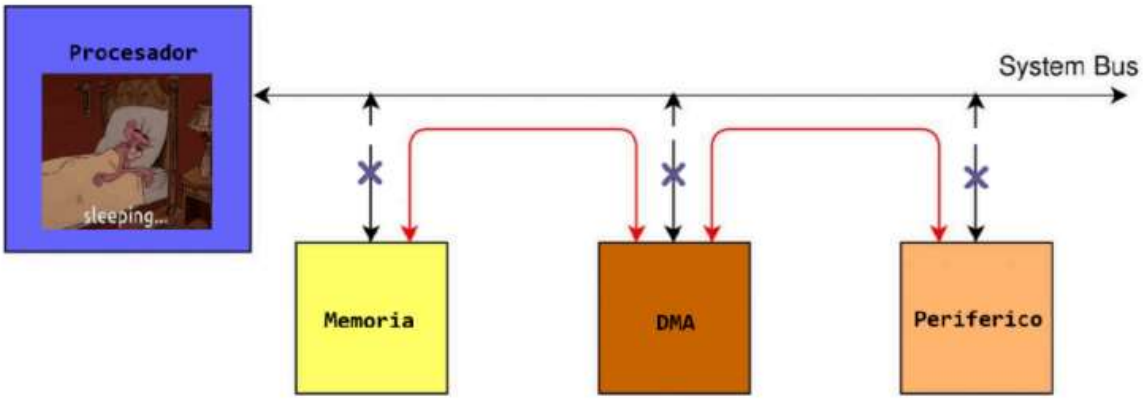
Transformaciones de la Señal de audio entrante

La señal analógica inicial es captada por un micrófono (**Módulo Ky-037**) y convertida a su equivalente digital mediante el **convertidor analógico a digital (ADC)** de la placa. Esta conversión genera una representación digital que se almacena directamente en la memoria para su posterior procesamiento. Para evitar interferencias con los procesos críticos del procesador, se utiliza el **DMA (Direct Memory Access)**, que transfiere los datos digitales desde la memoria al módulo **DAC.** Este módulo, a su vez, convierte los datos digitales nuevamente en una señal analógica, que se reproduce a través de un altavoz.La señal final es una representación bastante fiel de la original, captada por el micrófono. Sin embargo, es importante señalar que tanto el **ADC** como el **DAC** cuentan con un número limitado de bits de conversión, lo cual puede afectar la claridad de las señales de audio. A pesar de ello, esta limitación no impacta en la comprensión de la señal reproducida.

El siguiente esquemático representa la transformación de la señal a través de la información:



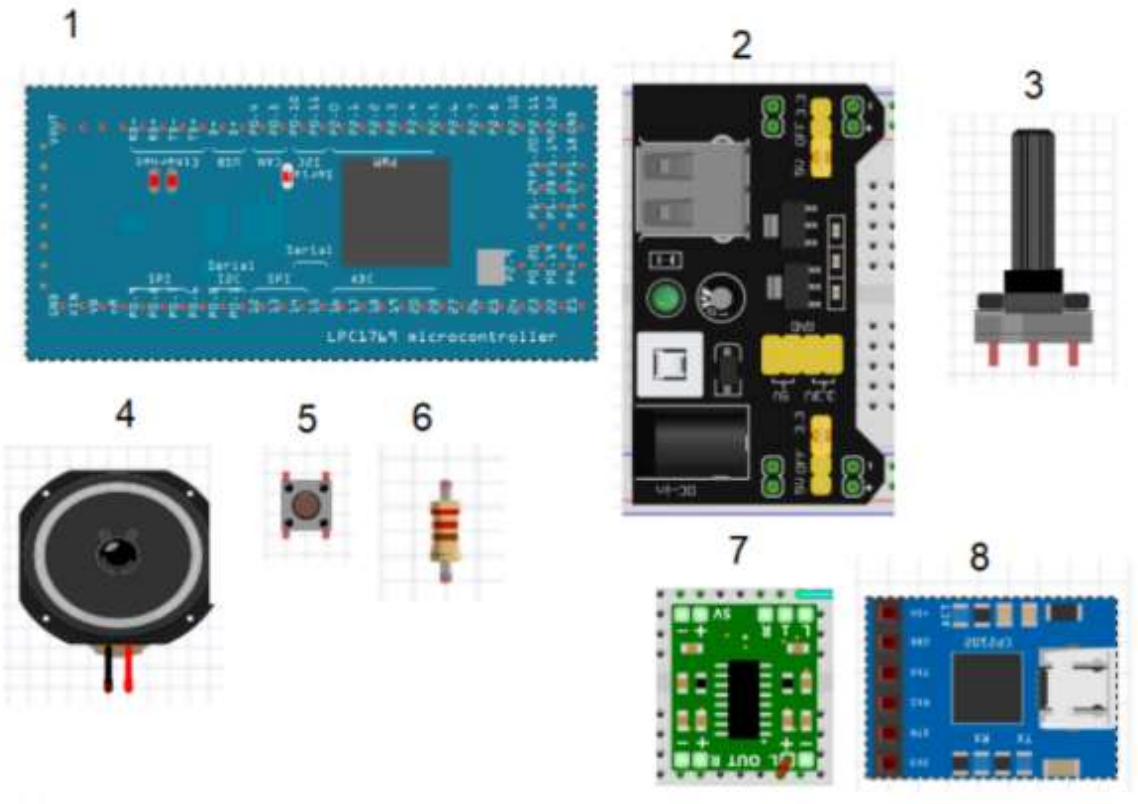
El esquemático a continuación representa el accionar del DMA en el trabajo:



Básicamente, el **DMA (Direct Memory Access)** opera a través de un controlador dedicado que gestiona las transferencias de datos entre los dispositivos periféricos y la memoria. Cuando se inicia una transferencia, el controlador **DMA** toma el control del bus de datos y realiza la transferencia directamente entre el periférico y la memoria, eliminando la necesidad de intervención de la CPU. Este proceso permite que la CPU permanece libre para realizar otras tareas críticas, mejorando significativamente el rendimiento del sistema.

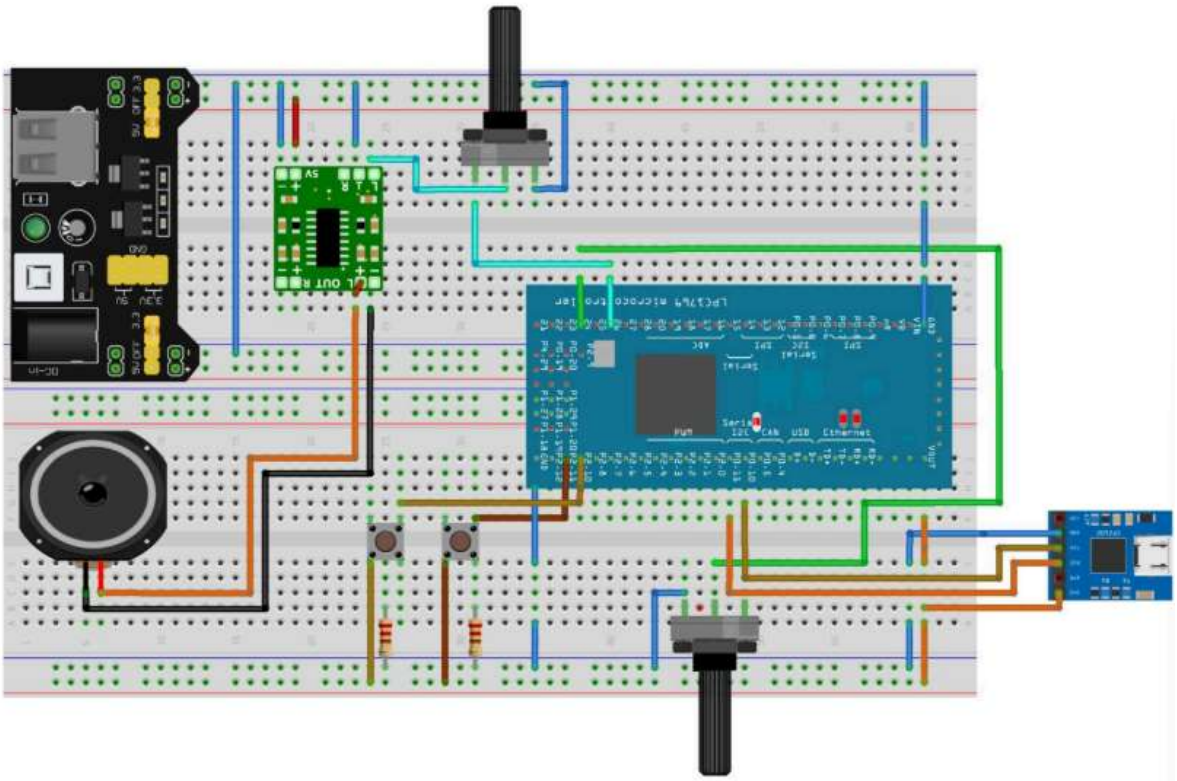
Desarrollo

A continuación se enumeran los componentes utilizados:



Se nombran a continuación:
1. LPC1769
2. Fuente POW-BREADBOARD 9v a 5v
3. Dos Potenciómetros: uno varía la frecuencia de 1 KHz y otro controla el volumen del parlante de
10 kHz
4. Parlante (8 ohm 1 watt)
5. Pulsador
6. Resistencias 10 KΩ
7. Amplificador PAM8403
8. TTL-UART Cp2102

Montado del circuito



Cálculos

Cantidad de Datos a Guardar

La **SRAM** del sistema tiene una capacidad máxima de **32 KB.** En el circuito, los datos grabados por el micrófono, tras ser convertidos a digital, se almacenan en un arreglo llamado **listADC**, que puede contener hasta **12,000 datos** de tipo **uint16_t.** Esto ocupa **24 KB** de la memoria disponible. Si se decide almacenar los datos en una variable declarada como **const**, el compilador los tratará como variables de solo lectura (**read-only**), lo que permite aprovechar mejor la memoria y almacenar una mayor cantidad de datos.

TimeOut

El **TimeOut** es el valor que determina el intervalo en el que el controlador de **DMA** interrumpe para enviar las muestras al pin **AOUT** del **DAC.** En este caso, dicho pin está conectado a un parlante.

La fórmula para calcular el **TimeOut** es:

$$\text{TimeOut} = \frac{F_{cclk}}{F_{\text{muestreo}} \times N}$$

Donde:

- **FcclkF:** Frecuencia del reloj del sistema.
- **Fmuestreo** Frecuencia de muestreo.
- **N:** Número de datos procesados por ciclo de interrupción.

De esa forma variamos con un potenciómetro el TimeOut.

UART

Este trabajo práctico emplea la comunicación serial **UART** en dirección **PC -> LPC**. Para la transmisión de datos, primero se convierte un archivo **MP3/WAV** a valores decimales utilizando las herramientas **"Audacity"** y **.EncoderAudio**. Luego, un **script en Python** parsea estos valores y los envía, uno por uno, al sistema.

La configuración del **UART** se realizó utilizando principalmente los valores predeterminados de los controladores **CMSIS** empleados durante la cursada. Entre las configuraciones predeterminadas, se destaca:

- **Baud rate**: **9600 bps**.
- **Configuración FIFO** con **DMA deshabilitado**, lo que significa que el sistema genera una interrupción cada vez que se recibe un dato.
- **Reseteo del buffer** de transmisión y recepción en cada ciclo de operación.

Además, las **banderas de error** utilizadas en el **handler** son las siguientes:

- UART_LSR _OE : Overrun error, un nuevo caracter se incluyo en una FIFO llena.
- UART_LSR _PE : Parity error.
- UART_LSR _FE : Framing error. Cuando el bit de stop es un cero.
- UART_LSR _BI : Break interrupt, Cuando el caracter completo de la transmisión es todo cero.
- UART_LSR _RXFE: Cualquier de los anteriores errores para la FIFO de Rx.

Como primera instancia en el handler testeamos si alguna de estas banderas es 1, si alguna está activada se queda en un loop infinito a modo de error, si no se toma el valor.

Código

```
#include "LPC17xx.h"
#include "lpc_types.h"
#include "lpc17xx_adc.h"
#include "lpc17xx_dac.h"
#include "lpc17xx_gpdma.h"
#include "lpc17xx_pinsel.h"
#include "lpc17xx_exti.h"
```

```
#include "lpc17xx_uart.h"

#define ADC_RATE    8000                // A mayor ADCRATE mayor
fidelidad de sonido.
#define LISTSIZE    12000               // No superar los 15k muestras
por que se llena la SRAM 32kB.
#define TIMEOUT     7000                // Arranca por default en un
valor, pero el potenciometro lo varia durante la ejecucion.
#define NUM_LISTS   3                   // Cada lista es de 4095
valores.

void configADC(void);
void configDAC(void);
void configGPIO(void);
void configEINT0(void);
void configUART(void);
void configEINT1(void);
void configDMA(__IO uint16_t listADC[]);
void configNVIC(void);

uint32_t map(uint32_t x, uint32_t in_min, uint32_t in_max, uint32_t
out_min, uint32_t out_max);
void cleanListADC(void);
void moveListDAC(void);
void buttonDebounce(void);

/* Esta lista guarda el sonido grabado por el microfono. */
__IO uint16_t listADC[LISTSIZE] = {0};
__IO uint32_t *samples_count = (__IO uint32_t *)0x2007C000;     //
Contador de muestras para la lista del ADC.

/* Variables para la comunicacion UART. */
uint8_t info[1]     = "";
uint32_t count_UART = 0;

/* Variable global para switchear de canal del ADC entre mic y
potenciometro. */
uint8_t RECORDING   = 0;

GPDMA_LLI_Type LLI_Array[NUM_LISTS];
GPDMA_Channel_CFG_Type dmaCFG;
```

```
/*
-------------------------------------------------------------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
 *
-----------------------------------------------MAIN----------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
 */
int main()
{
    configGPIO();
    configEINT0();
    configEINT1();
    configADC();
    configDAC();
    configUART();
    configNVIC();

    while(1)
    {
        // idle...
    }

    return 0;
}




/*
-------------------------------------------------------------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
```

```
 *
-----------------------------------------FUNCTIONS--------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
 */
void cleanListADC(void)
{
    /* Rellenar con ceros la lista del ADC. */

    for (uint32_t i = 0; i < LISTSIZE; i++)
    {
        listADC[i] = 0;
    }
}


void moveListDAC(void)
{
    /* Desplazamos los valor de la lista 6 lugares, 4 para el DAC y 2
mas para recortar los LSB. */

    for (uint32_t i = 0; i < LISTSIZE; i++)
    {
        listADC[i] = listADC[i]<<6;
    }
    return;
}


void buttonDebounce(void)
{
    /* Delay para antirrebote del botones.
     * Se deberia hacer con un TIMER, no de esta manera.
     */

    for (uint32_t i = 0; i < 50000; i++){}
}
```

```
uint32_t map(uint32_t x, uint32_t in_min, uint32_t in_max, uint32_t
out_min, uint32_t out_max)
{
    /* Convierte el valor recibido a un valor correspondiente dentro de
una escala MIN-MAX dada. */
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) +
out_min;
}


/*
------------------------------------------------------------------------
-------------------------
 *
------------------------------------------------------------------------
-------------------------
 *
-----------------------------------------CONFIGS---------------------
-------------------------
 *
------------------------------------------------------------------------
-------------------------
 *
------------------------------------------------------------------------
-------------------------
 */
void configGPIO(void)
{
    /* Set P0.23 AD0.0 */
    PINSEL_CFG_Type pinCFG;
    pinCFG.Funcnum      = PINSEL_FUNC_1;
    pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
    pinCFG.Pinmode      = PINSEL_PINMODE_TRISTATE;
    pinCFG.Pinnum       = PINSEL_PIN_23;
    pinCFG.Portnum      = PINSEL_PORT_0;
    PINSEL_ConfigPin(&pinCFG);

    /* Set P0.24 AD0.1 */
    pinCFG.Funcnum      = PINSEL_FUNC_1;
    pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
    pinCFG.Pinmode      = PINSEL_PINMODE_TRISTATE;
    pinCFG.Pinnum       = PINSEL_PIN_24;
```

```
    pinCFG.Portnum      = PINSEL_PORT_0;
    PINSEL_ConfigPin(&pinCFG);

    /* Set P2.10 EINT0*/
    pinCFG.Funcnum      = PINSEL_FUNC_1;
    pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
    pinCFG.Pinmode      = PINSEL_PINMODE_PULLDOWN;
    pinCFG.Pinnum       = PINSEL_PIN_10;
    pinCFG.Portnum      = PINSEL_PORT_2;
    PINSEL_ConfigPin(&pinCFG);

    /* Set P2.11 EINT1 */
    pinCFG.Funcnum      = PINSEL_FUNC_1;
    pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
    pinCFG.Pinmode      = PINSEL_PINMODE_PULLDOWN;
    pinCFG.Pinnum       = PINSEL_PIN_11;
    pinCFG.Portnum      = PINSEL_PORT_2;
    PINSEL_ConfigPin(&pinCFG);

    /* Set P0.26 AD0.0 */
    pinCFG.Funcnum      = PINSEL_FUNC_2;
    pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
    pinCFG.Pinmode      = PINSEL_PINMODE_TRISTATE;
    pinCFG.Pinnum       = PINSEL_PIN_26;
    pinCFG.Portnum      = PINSEL_PORT_0;
    PINSEL_ConfigPin(&pinCFG);

    /* P.022 Output LED */
    LPC_GPIO0->FIODIR   |= (1<<22);     // Led Rojo
    LPC_GPIO3->FIODIR   |= (1<<25);     // Led Verde
    LPC_GPIO3->FIODIR   |= (1<<26);     // Led Azul
    LPC_GPIO0->FIOSET   |= (1<<22);     // Apaga el led rojo.
    LPC_GPIO3->FIOSET   |= (1<<25);     // Apaga el led verde.
    LPC_GPIO3->FIOSET   |= (1<<26);     // Apaga el led azul.

    /* Configuracion pin de Tx y Rx */
    pinCFG.Funcnum      = 1;
    pinCFG.OpenDrain    = 0;
    pinCFG.Pinmode      = 0;
    pinCFG.Pinnum       = 10;
    pinCFG.Portnum      = 0;
    PINSEL_ConfigPin(&pinCFG);  // Tx
    pinCFG.Pinnum       = 11;
```

```
    PINSEL_ConfigPin(&pinCFG);  // Rx
    return;
}


void configADC(void)
{
    /* El ADC se utiliza en modo burst para tener el maximo de
resolucion.
     * La interrupcion se activa en configNVIC() y comienza apagada ya
que se prende con el pulsador en EINT0.
     *
     * Tenemos dos canales de interrupcion, uno para el mic y otro para
el potenciometro, pero
     * solamente activamos la interrupcion del microfono, ya que el
potenciometro va a depender del estado de
     * la variable RECORDING.
     */

    ADC_Init(LPC_ADC, ADC_RATE);
    ADC_StartCmd(LPC_ADC, ADC_START_CONTINUOUS);
    ADC_ChannelCmd(LPC_ADC, 0, ENABLE);
    ADC_ChannelCmd(LPC_ADC, 1, ENABLE);
    ADC_BurstCmd(LPC_ADC, ENABLE);
    ADC_IntConfig(LPC_ADC, ADC_ADINTEN0, ENABLE);
}


void configDAC(void)
{
    DAC_CONVERTER_CFG_Type dacCFG;
    dacCFG.CNT_ENA = SET;
    dacCFG.DMA_ENA = SET;

    DAC_SetDMATimeOut(LPC_DAC, TIMEOUT);                /* REVISAR
CALCULO DE TIMEOUT. */
    DAC_ConfigDAConverterControl(LPC_DAC, &dacCFG);
    DAC_Init(LPC_DAC);
}


void configDMA(__IO uint16_t listADC[])
{
```

```
    for (int i = 0; i < NUM_LISTS; i++)
    {
        LLI_Array[i].DstAddr = (uint32_t) &(LPC_DAC->DACR);
        LLI_Array[i].SrcAddr = (uint32_t)(listADC + i * 4095);

        if (i == (NUM_LISTS - 1))
        {
            LLI_Array[i].NextLLI = (uint32_t)&LLI_Array[0];
        }
        else
        {
            LLI_Array[i].NextLLI = (uint32_t)&LLI_Array[i + 1];
        }

        LLI_Array[i].Control = 4095
                             | (1 << 18)    // source width 16 bit
                             | (1 << 22)    // dest width = word 32
bits
                             | (1 << 26)    // source increment
                             ;
    }

    dmaCFG.ChannelNum           = 0;
    dmaCFG.TransferSize         = 4095;
    dmaCFG.TransferWidth        = 0;
    dmaCFG.TransferType         = GPDMA_TRANSFERTYPE_M2P;
    dmaCFG.SrcConn              = 0;
    dmaCFG.DstConn              = GPDMA_CONN_DAC;
    dmaCFG.SrcMemAddr           = (uint32_t) listADC;
    dmaCFG.DstMemAddr           = 0;
    dmaCFG.DMALLI               = (uint32_t) &LLI_Array[0];

    GPDMA_Init();
    GPDMA_Setup(&dmaCFG);
    GPDMA_ChannelCmd(0, ENABLE);
    return;
}


void configEINT0(void)
{
    /* Interrupcion para inicializar la grabacion y el ADC.
     * Las interrupciones se activan en configNVIC()
```

```
     */

    EXTI_InitTypeDef exti;
    exti.EXTI_Mode      = EXTI_MODE_EDGE_SENSITIVE;
    exti.EXTI_polarity  = EXTI_POLARITY_HIGH_ACTIVE_OR_RISING_EDGE;
    exti.EXTI_Line      = EXTI_EINT0;

    EXTI_Config(&exti);
}


void configEINT1(void)
{
    /* Interrupcion para poner play/pausa el sonido.
     * Las interrupciones se activan en configNVIC()
     */

    EXTI_InitTypeDef exti;
    exti.EXTI_Mode      = EXTI_MODE_EDGE_SENSITIVE;
    exti.EXTI_polarity  = EXTI_POLARITY_HIGH_ACTIVE_OR_RISING_EDGE;
    exti.EXTI_Line      = EXTI_EINT1;

    EXTI_Config(&exti);
}


void configNVIC(void)
{
    LPC_ADC->ADGDR &= LPC_ADC->ADGDR;
    NVIC_EnableIRQ(ADC_IRQn);

    EXTI_ClearEXTIFlag(EXTI_EINT0);
    NVIC_EnableIRQ(EINT0_IRQn);

    EXTI_ClearEXTIFlag(EXTI_EINT1);
    NVIC_EnableIRQ(EINT1_IRQn);

    NVIC_EnableIRQ(UART2_IRQn);

    GPDMA_ChannelCmd(0, DISABLE);
}
```

```
void configUART(void)
{
    UART_CFG_Type      UARTConfigStruct;
    UART_FIFO_CFG_Type UARTFIFOConfigStruct;
    // Configuracion por defecto 9600 baud-rate.
    UART_ConfigStructInit(&UARTConfigStruct);
    // Inicializa periferico
    UART_Init(LPC_UART2, &UARTConfigStruct);
    // Inicializa FIFO
    UART_FIFOConfigStructInit(&UARTFIFOConfigStruct);
    UART_FIFOConfig(LPC_UART2, &UARTFIFOConfigStruct);
    // Habilita interrupcion por el RX del UART
    UART_IntConfig(LPC_UART2, UART_INTCFG_RBR, ENABLE);
    // Habilita interrupcion por el estado de la linea UART
    UART_IntConfig(LPC_UART2, UART_INTCFG_RLS, ENABLE);
}




/*
-------------------------------------------------------------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
 *
-----------------------------------------HANDLERS---------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
 *
-------------------------------------------------------------------------
-------------------------
 */
void ADC_IRQHandler(void)
{
    static uint32_t ADCVAL      = 0;
    static uint32_t ADCVALMAP   = 0;

    /* Tomamos una muestra del ADC y la guardamos en el array sin
superar el limite de muestras. */
```

```
    if (RECORDING > 0)
    {
        if (*samples_count <= LISTSIZE)
        {
            /* Comenzamos a grabar un audio y guardarlo en el array. */
            LPC_GPIO3->FIOCLR |= (1<<26);   // Prende el led azul.
            listADC[*samples_count] = ((LPC_ADC->ADDR0)>>6) & 0x3FF;
            (*samples_count)++;
        }
        else
        {
            LPC_GPIO0->FIOSET |= (1<<22);   // Apaga el led rojo.
            LPC_GPIO3->FIOSET |= (1<<25);   // Apaga el led verde.
            LPC_GPIO3->FIOSET |= (1<<26);   // Apaga el led azul.
            *samples_count = 0;
            RECORDING = 0;
            moveListDAC();
        }
    }
    else if (RECORDING == 0)
    {
        /* Si NO estamos grabando variamos la frecuencia de salida del
DAC. */
        ADCVAL      = ((LPC_ADC->ADDR1)>>6) & 0x3FF;
        ADCVALMAP   = map(ADCVAL, 0, 1024, 5000, 20000);
        DAC_SetDMATimeOut(LPC_DAC, ADCVALMAP);
    }

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


void EINT0_IRQHandler(void)
{
    /* Comenzamos a grabar un sonido por el ADC.
     * Al llenarse el array de valores se detiene automaticamente la
grabacion a
     * la espera de poner en play el sonido.
     */

    buttonDebounce();
```

```
    RECORDING = 1;

    LPC_GPIO0->FIOSET |= (1<<22);   // Apaga el led rojo.
    LPC_GPIO3->FIOSET |= (1<<25);   // Apaga el led verde.
    LPC_GPIO3->FIOSET |= (1<<26);   // Apaga el led azul.
    GPDMA_ChannelCmd(0, DISABLE);
    *samples_count = 0;
    cleanListADC();

    NVIC_EnableIRQ(ADC_IRQn);
    EXTI_ClearEXTIFlag(EXTI_EINT0);
}

void EINT1_IRQHandler(void)
{
    /* Si esta reproduciendo sonido, deshabilita el canal y baja a 0 la
salida.
     * Si esta en pausa, pone en play la reproduccion de sonido.
     */
    static uint8_t PLAY = 0;

    buttonDebounce();

    if (PLAY > 0)
    {
        LPC_GPIO3->FIOSET |= (1<<25);   // Apaga el led verde.
        LPC_GPIO3->FIOSET |= (1<<26);   // Apaga el led azul.
        LPC_GPIO0->FIOCLR |= (1<<22);   // Prende el led rojo.
        GPDMA_ChannelCmd(0, DISABLE);
        DAC_UpdateValue(LPC_DAC, 0);
        PLAY = 0;
    }
    else
    {
        LPC_GPIO0->FIOSET |= (1<<22);   // Apaga el led rojo.
        LPC_GPIO3->FIOSET |= (1<<26);   // Apaga el led azul.
        LPC_GPIO3->FIOCLR |= (1<<25);   // Prende el led verde.
        configDMA(listADC);
        PLAY = 1;
    }

    EXTI_ClearEXTIFlag(EXTI_EINT1);
}
```

```
void UART2_IRQHandler(void)
{
    uint32_t intsrc, tmp, tmp1;

    // Determina la fuente de interrupcion
    intsrc = UART_GetIntId(LPC_UART2);
    tmp = intsrc & UART_IIR_INTID_MASK;

    // Evalua Line Status - Received-line status.
    if (tmp == UART_IIR_INTID_RLS)
    {
        tmp1 = UART_GetLineStatus(LPC_UART2);
        tmp1 &= (UART_LSR_OE | UART_LSR_PE | UART_LSR_FE | UART_LSR_BI
| UART_LSR_RXFE);
        if (tmp1)
        {
            while(1){};     /* ingresa a un loop infinito si hay error
*/
        }
    }

    // Receive Data Available or Character time-out
    if ((tmp == UART_IIR_INTID_RDA) || (tmp == UART_IIR_INTID_CTI))
    {
        UART_Receive(LPC_UART2, info, sizeof(info), NONE_BLOCKING);
    }

    /* A veces el UART tiene un bug que manda 0 de por medio por eso el
condicional. */
    if ((count_UART < LISTSIZE) & (info[0] != 0))
    {
        listADC[count_UART] = (info[0]<<6);
        count_UART++;
    }

    if (count_UART >= LISTSIZE)
    {
        count_UART = 0;
    }

    return;
}
```

Conclusión

En conclusión, el sistema diseñado ofrece una funcionalidad eficiente y versátil para la captura, almacenamiento y reproducción de señales de audio. La activación del botón **'Record'** permite la adquisición directa de la señal analógica del micrófono, la cual se almacena en la memoria sin necesidad de utilizar **DMA**, simplificando el proceso y optimizando los recursos. La gestión de la memoria se realiza de manera efectiva, ya que esta se reinicia automáticamente al grabar un nuevo sonido, asegurando un uso eficiente del espacio de almacenamiento.

Por otro lado, la reproducción de la señal se inicia mediante el botón **'Play/Stop'**, lo que desencadena la extracción de datos de la memoria y su envío al **DAC** a través de **DMA**. La frecuencia del sonido se ajusta dinámicamente mediante un **potenciómetro**, lo que proporciona flexibilidad en la reproducción del audio. La salida analógica resultante se dirige al pin designado (**P0.26**), ofreciendo una interfaz clara y accesible para la reproducción de sonidos.

Además, el sistema incluye una funcionalidad adicional de comunicación a través del **UART**, lo que permite la transferencia de secuencias de sonidos en forma de valores decimales.

En conjunto, la integración de estas características proporciona un sistema completo y flexible para la captura, almacenamiento y reproducción de audio, abriendo posibilidades de comunicación adicional a través del **UART**.