REPRODUCTOR DE RINGTONES

Trabajo practico n°3

Universidad Nacional de La Plata
Facultad de Ingeniería
Circuito Digitales y Microcontroladores

Francisco Pérez
Gallo Francisco, Molinuevo

Índice

1. Introducción

Se debe implementar un reproductor de ringtones y la comunicación serie UART para poder comunicarse con la terminal. De esta forma se mandarán comandos y dependiendo del dato recibido, realizar cierta acción sobre el reproductor. Se debe verificar que tipo de comando es y, además, que sea válido. Caso contrario, que el usuario disponga de un feedback que cometió un error.

Se utilizará una arquitectura de tipo Background/Foreground. También se utilizarán 2 timer, el TIMER0 para la temporización del proyecto y el TIMER1 para la generación de señal la cual esa vinculada a la reproducción de un sonido.

Antes de comenzar, se debe dar a conocer que se dio a entender que los comandos funcionaban de otra manera, es decir, se realizó el trabajo definiendo que el comando STOP detiene la canción actual sonando y no que vuelve a empezar a tocar toda la librería de canciones de cero.

2. Temporizadores:

2.1 Introducción:

El proyecto hará uso, como se mencionó, de 2 temporizadores del MCU. Siendo estos el TIMER1 y TIMER0

2.2 TIMER1:

Este periférico es un contador/timer de 16 bits el cual tiene varios modos de operación. La utilización de este es para generar frecuencias especificas el cual el parlante las convertirá en sonido. Esto es posible ya que, al ser de 16 bits, el temporizador es capaz de generar 2^{16}=65536 frecuencias posibles en el modo CTC. Ya que la frecuencia generada esta relacionada al valor de comparación en el registro (registro de 16 bits).

2.2.1 Configuración:

Se modificarán los registros de configuración del timer. Se utilizará los 16MHz como frecuencia de entrada. Luego modo CTC para poder generar una interrupción cuando alcanza el valor de comparación definida y luego que la señal generada salga por un pin (PB1).

```
TCCR1A|=(1<<COM1A0);// Configuro Timer1 para clk con prescaler P=1,
modo CTC y salida por pin
TCCR1B|=(1<<WGM12)|(1<<CS10);
DDRB|=(1<<PINB1); // El PIN1 del PORTB será el pin de salida
```

La cantidad de señales que se pueden generar dependen de la cantidad de valores en el registro de comparación ya que, la *frecuencia*, el *preescalador* son fijos y lo único que variamos es el *OCRnA*

$$f_{OCnA} = \frac{f_{clk\_I/0}}{2.N(1 + OCRnA)}$$

Luego, una librería que será explicada posteriormente, ira modificando este registro y generará la señal correspondiente en ese momento.

2.3 TIMER0:

La librería que provee la catedra para generar un ringtone necesita una temporización. Para esto utilizaremos este timer de 8 bits ya usado en trabajos anteriores.

2.3.1 Utilización modo CTC

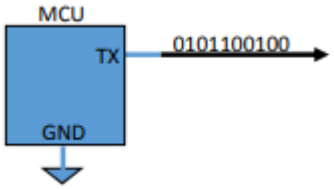
Se configurará en modo CTC para poder generar una interrupción cada 1 ms. La librería la utiliza para configurar duración de un sonido y demás.

```
// Configuro una interrupción cada 1 mseg
      OCR0A = 248;      //124 para 8MHz y 248 para 16MHz
      TCCR0A = (1<<WGM01); // Modo CTC, clock interno, prescalador 64
      TCCR0B = (1<<CS01)|(1<<CS00)// Modo CTC, clock interno, prescalador 64
      TIMSK0 = (1<<OCIE0A);// Habilito Timer 0 en modo de interrupción de
      comparación
```

3. Comunicación serie:

3.1 Introducción

Una transmisión o comunicación serie es aquella donde los datos enviados son en forma de paquete de varios bits, un bit a la vez, por un mismo canal de comunicación



3.1.1 Conceptos básicos:

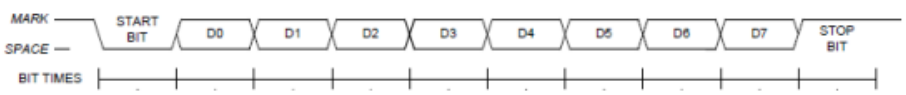
Las principales características de este tipo de comunicación son:

- Tiempo de bit (Tb): duración de un bit en la transmisión
- Tasa de transferencia (1/Tb): número de bits por unidad de tiempo [bps]. Otra forma de llamarlo es “baud-rate” (símbolos por segundo)
- Overhead: el overhead son bits adicionales que se le agregan al dato en la trasmisión para hacer esta más confiable. Ej: bit de paridad, checksum.
- Ancho de banda: número total de bits de **información** por unidad de tiempo, sin tener en cuenta el Overhead.

Hay varias formas de realizar la comunicación entre dos dispositivos, existe el full-duplex, half-duplex y simplex
La comunicación full-duplex donde ambos dispositivos son transmisores y receptores simultáneamente. Requiere hardware separado y dos canales de comunicación
Por otro lado, half-duplex es una comunicación bidireccional pero no simultanea ya que comparten mismo canal de comunicación (existencia de colisiones)
Simplex es comunicación hacia un solo sentido

3.1.2 Forma de trama:

La transmisión además de mandar datos, debe mandar información para sincronizarse (bit de start y bit de stop)

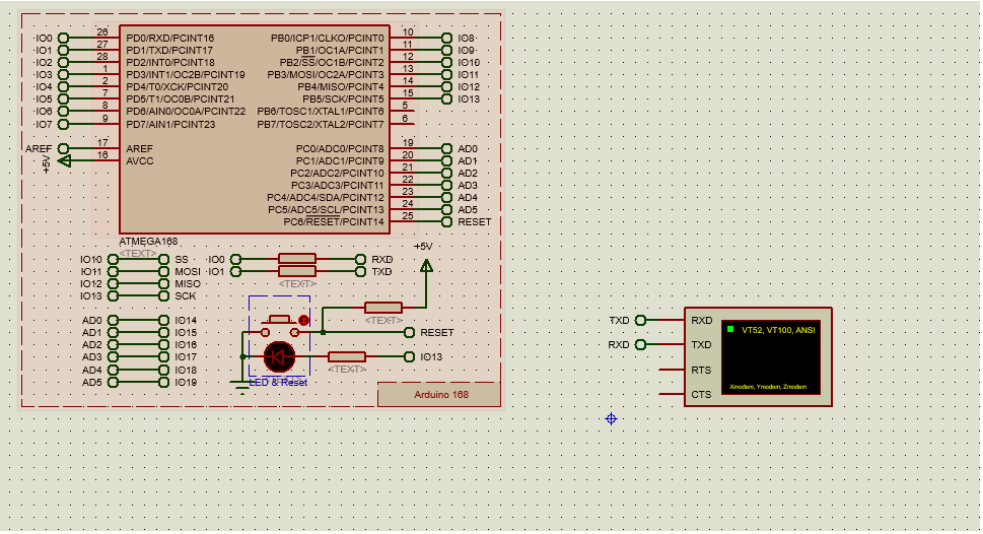


3.3 UART:

La UART es un periférico integrado en el MCU que permite el tipo de transmisión descripta anteriormente con otros dispositivos como una PC, módems u otros MCU.

3.3.2 Comunicación entre 2 dispositivos:

Se necesita la comunicación del MCU con una terminal de comandos para así el usuario poder utilizar unos comandos predefinidos. Estos comandos eran recibidos por la pulsación del teclado. La confirmación de este era apretando la letra ENTER. (ver conexiado-1)



Conexionado-1 conexión entre transmisor y receptor

Esta comunicación es serial, asincrónica, full duplex y 8N1. Es decir que, los 2 podían actuar como transmisor o receptor, dependiendo la necesidad y que, la forma de trama era 8 bits con uno adicional de STOP.

3.3.2 Periféricos:

3.3.2.1 Generador de reloj:

Este utiliza un divisor de reloj para establecer la tasa de transmisión. Para configurar se utilizan los registros **UBBRH** (USART Baud Rate Register High) y **UBBRL** (USART Baud Rate Register Low). Encargados de determinar cómo se divide la frecuencia de reloj del microcontrolador para obtener una tasa de transmisión conocida deseada.
Para calcular el valor de estos registros se utiliza la siguiente formula:

$$UBBR = \left(\frac{f_{osc}}{(16 * BAUD)} - 1\right)$$

En este proyecto se utilizará un baudrate estándar de 9600, por lo tanto:

$$UBBR = \left(\frac{16Mhz}{(16 * 9600)} - 1\right) = 103$$

Como los valores de UBBR son discretos, se comete un error en la cuenta, siendo este:

$$Error\ cometido = \frac{\left(\frac{16Mhz}{(16 * 9600)} - 1\right)}{9600} \cong 0.2\%$$

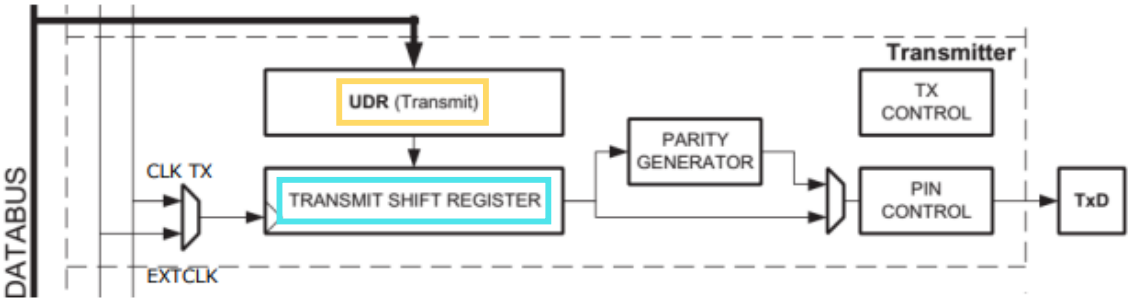
Para que una comunicación sea confiable, el error cometido debe ser menor al 5% por lo que estamos en un buen porcentaje de error.
Es importante tener en cuenta que el transmisor y receptor deben tener configurada la misma velocidad de transmisión para una correcta comunicación.

3.3.2.2 Transmisor:

La transmisión es un conversor de datos paralelo a serie con un reloj derivado del reloj de MCU

Para que se pueda transmitir un dato, se debe habilitar el transmisor (**TXEN**=1) y configurarse de forma apropiada

Donde el **UDR** (USART Data Register) es para cargar el dato a transmitir, funcionando como buffer y luego se transfiere al **Transmit Shift Register.**



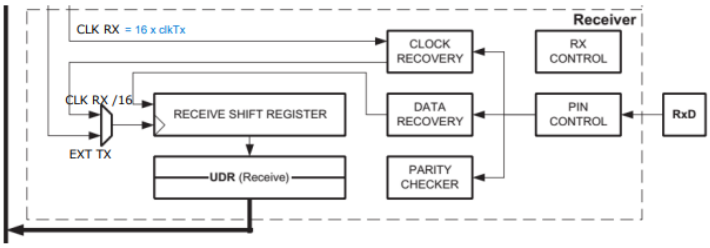
El Transmit Shif Register es un registro de desplazamiento utilizado para almacenar los datos que se van a transmitir. Los datos son cargados en este registro y se envían uno por uno

Si el registro UDR está vacío (**UDRE** = 1) el usuario puede cargar un dato para transmitir.

El flag **TXC** se activa (**TXC** = 1) cuando el transmisor haya completado la transmisión y tanto el **UDR** y el **TSS** estén vacíos. (Ambos flags se borran automáticamente con la escritura de un dato en el **UDR**)

3.3.2.3 Receptor:

El receptor en cambio es un conversor de datos serie a paralelo, sincronizado con el reloj derivado del reloj del MCU (modo asincrónico) o el reloj que recibe del transmisor remoto (modo sincrónico).



Para recibir un dato debe habilitarse el receptor (RXEN = 1) y ser configurado apropiadamente

El clock del receptor es 16 veces más rápido que el del receptor para poder consultar varias veces el estado de un bit transmitido

Una vez detectado el bit de comienzo, los datos son muestrados e introducidos al registro de desplazamiento a medida que son decodificados y hasta detectar el bit de parada o STOP.

Luego de que se complete la recepción, el contenido dentro del TSS se transfiere al UDR y se activa el flag de “dato recibido” **RXC**. Es posible configurar para que este flag genere una solicitud de interrupción si se habilita **RXCIE**.

Luego de esto, un nuevo dato puede ser recibido mientras el anterior se encuentra almacenado en el UDR. Constituyendo así un mecanismo de doble buffer. Si se completa la segunda recepción, el dato anterior en el UDR es reemplazado y perdido.

3.4 Configuración:

En este proyecto se optó como se dijo, de una tasa de transmisión de 9600bps, además, de que la transmisión se hará mediante interrupciones para que la transmisión no sea bloqueante. La catedra proveyó una librería para una sintaxis más clara. Por ejemplo, las funciones

```
void SerialPort_Init(uint8_t);
void SerialPort_TX_Enable(void);
void SerialPort_TX_Interrupt_Enable(void);
void SerialPort_TX_Interrupt_Disable(void);
void SerialPort_RX_Enable(void);
void SerialPort_RX_Interrupt_Enable(void);
void SerialPort_RX_Enable(void);
void SerialPort_RX_Interrupt_Enable(void);
```

En donde cada una de estas modifica los valores de los registros para la configuración. El tipo de configuración es trivial por el nombre de la función.
Estos registros son el **UCSRA, UCSRB y UCSRC**

3.5 Implementación:

Se construyó una biblioteca de funciones para el periférico UART encargado de enviar y recibir mensajes de una cantidad de bytes determinada.
Es un problema denominado Productor/Consumidor y se basa en la estructura FIFO (First In – First Out)
Se debe tener en cuenta el orden de ejecución, por ejemplo, el caso de que primero un dato debe recibirse para poder transmitirlo. Esto se denomina sincronización y se implementara mediante flags.
Como luego es explicado, en arquitectura background/foreground una de las tareas es un handler de interrupción y la otra tarea de segundo plano. En este caso tendremos dos handler, una para la recepción y otra para la transmisión.

Por otro lado, se utilizará un buffer para almacenar lo que es recibido y, otro buffer para poder depositar lo que se necesita transmitir. Un buffer es una estructura de tamaño fijo, residente en RAM. Por lo cual aloja temporalmente un conjunto de datos.
Cada buffer tendrá un espacio de 50 bytes.

Este driver proveerá funciones para poder mandar un carácter, mandar un string, un numero entero de un digito, de almacenar una cadena y obtener la cadena (si es que la hay) que se recibió.

Se empezará por la función que permite mandar un string a transmitir como parámetro de esta.

```
void UART_Send_String_To_Transmit(char* string_pointer)
```

El driver, en este caso maneja un flag para que, si anteriormente se estaba transmitiendo un dato, esperar a que el dato entero se envíe, es decir, si se estaba mandando un string antes de que se llame a la función, esta no podrá transmitir hasta que termine (ver Pseudocodigo 1).

```
Mientras un dato no haya terminado de transmitirse
    Esperar
Establecer el flag que indica que se está transmitiendo un nuevo dato
Obtener la longitud de la cadena
Copiar la cadena en el buffer de transmisión
Habilito la interrupción de transmisión
```

Pseudocodigo 1

Luego, la función para mandar un digito (entero) está dada como:

```
void UART_Send_Digit_To_Transmit(uint8_t number)
```

Donde antes de transmitir el número que se recibe como parámetro, hay que convertirlo a carácter ASCII y funciona similar a transmitir el string, guardo el numero convertido al buffer de transmisión, indico que tiene longitud 1 y habilito la interrupción de transmisión.
Así funciona también mandar un carácter, salteando el paso de la conversión ya que se recibe como parámetro un carácter.

```
void UART_Send_Char_To_Transmit(char caracter)
```

4. Arquitectura Background/Foreground.

Esta arquitectura se basa en la separación de las tareas en dos categorías, tal como su nombre indica, las de fondo y las de primer plano, donde las tareas de esta última tienen prioridad, y las restantes pueden interrumpir según sea necesario.

4.1 Introducción.

Para la comunicación UART, se utilizan interrupciones para la recepción y transmisión de datos de forma no bloqueante, lo que permite recibir datos sin detener la ejecución del programa principal. Se necesita también de un búfer para almacenar los datos recibidos, otro para enviarlos y un flag para chequear que se reciba un dato.

4.2 Implementación.

Flag de dato recibido. La función *UART_Update()* se encarga de actualizar la comunicación UART.

```
Si se recibió un dato
    Se copia el contenido del búfer de recepción al búfer de transmisión
    Se modifica la sintaxis para que se visualice correctamente en la
consola
    Indicar que la transmisión esta lista para comenzar
    Habilitar la interrupción de transmisión UART.
    Si se recibió un nuevo dato, pero la anterior transmisión no finalizo
       Esperar
   Levantar flag para que la MEF pueda conocer si se recibió nuevo dato
```

UART_Update()

Recepción. La interrupción ISR(USART_RX_vect) se produce cuando se recibe un dato.

```
Se lee el dato recibido del registro UDR0
Se guarda el dato recibido en el búfer de recepción
Si recibo un '\r' o se alcanzó el límite del búfer
    Se indica que se recibió un dato completo
    Se agrega '\0' al final del búfer
    Se reinicia el índice de escritura
Si no // La cadena no se ha completado aun
    Aumento en uno el índice de escritura
```

ISR(USART_RX_vect)

Transmisión. La interrupción ISR(USART_UDRE_vect) se produce cuando el registro de datos UDR0 está vacío y se puede enviar un nuevo dato.

```
Se carga el siguiente dato del búfer de transmisión para enviarlo

Se apunta al siguiente dato en el búfer de transmisión
Si la longitud de la cadena a transmitir es igual a 0 // Se han enviado
todos los datos del búfer
    Reiniciar los contadores
    Se deshabilita la interrupción de transmisión
Si no
    Se decrementa el contador de longitud de cadena
```

ISR(USART_UDRE_vect)

5. Reproductor de ringtones:

5.1 Introducción:

La catedra compartió una librería la cual es utilizada para generar y reproducir tonos de llamada (ringtones). RTTL es un lenguaje de texto que describe las notas, duración de las notas, la octava, el tempo y otros parámetros.

5.2 Funcionamiento:

Esta información se almacena en un arreglo de cadena de caracteres donde, a medida que pasa el tiempo, va leyendo este arreglo y generando el sonido correspondiente.

Un sonido está compuesto por frecuencias, la cual la librería almacena las necesarias para generar las notas. Estas notas (convertidas en frecuencias) son almacenadas en una matriz (ver código-1 matriz de frecuencias)

```
const unsigned int note[4][12] =
{   // C      C#    D     D#    E     F     F#    G     G#    A     A#    B
      {262,  277,  294,  311,  330,  349,  370,  392,  415,  440,  466,  494},
// 4ta octava
      {523,  554,  587,  622,  659,  698,  740,  784,  830,  880,  932,  988},
// 5ta octava
      {1047, 1109, 1175, 1244, 1319, 1397, 1480, 1568, 1660, 1760, 1865, 1976},
// 6ta octava
      {2093, 2218, 2349, 2489, 2637, 2794, 2960, 3136, 3320, 3520, 3728, 3951}
// 7ma octava
};
```

Código-1 matriz de frecuencias

Como se explicó en los temporizadores, específicamente el TIMER1, al ser de 16 bits tiene la posibilidad de generar un rango de frecuencias considerables. Solo es necesario modificar el registro OCRnA para poder generar la frecuencia buscada (ver ecuación-1)

$$f_{OCnA} = \frac{f_{clk\_I/0}}{2.N(1 + OCRnA)}$$

Ecuacion-1

Entonces, esta librería, ira leyendo el RTTL y modificando este registro para buscar la frecuencia deseada que indica la matriz de notas.

Como el registro es discreto entero, la frecuencia generada no será exactamente la misma, si no que se cometerá un error. Por ejemplo, si se requiere la frecuencia 2093:

$$OCR1A = \frac{16Mhz}{2093} - 1 = 7644{,}52$$

Al realizar un redondeo, se comete un error.

5.3 Modificación:

La librería provista por la catedra funcionaba correctamente, pero generaba problemas al implementarlo con la especificación requerida, ya que, una vez que se cargaba un ringtone, esta se reproducía hasta que termine, siendo bloqueante. Se necesitó modificar la función ***play_song(char *song)*** y dividirla en 2 funciones. Una de ellas denominada `void play_LoadSong(uint8_t selected_song)` la cual obtiene como parámetro el índice de la dirección del vector RTTL a tocar. Esta función carga y modifica las variables necesarias para la correcta reproducción de la canción como el tempo, la octava, etc.

Por otro lado, se tiene `uint8_t play_note(void)` que lee del vector de cadena de caracteres y toca la nota leída. En caso de que es la última nota, devuelve 0 para saber si se terminó la canción, caso contrario, devuelve 1.

También fue necesario eliminar un par de canciones ya que, el almacenamiento de estas ocupaba toda la memoria impidiendo así agregar más contenido a estas.

6. Interface.

Es una capa de abstracción que permite la comunicación con el usuario de forma clara y modularizada.

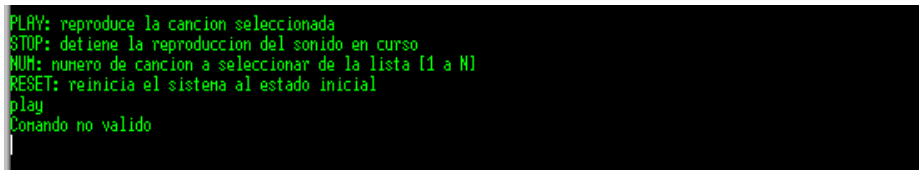
6.1 Introducción.

Esta interface se diseñó con el objetivo de mejorar la usabilidad y la interacción con el sistema. Se pueden mostrar instrucciones, opciones y mensajes de error de manera organizada, mejorando la usabilidad y facilitando la interacción con el sistema.

6.2 Implementación.

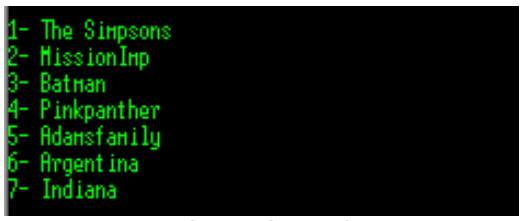
Se utiliza una estructura basada en mensajes predefinidos, almacenados en memoria de solo lectura (Memoria FLASH) para optimizar el uso de la misma y mejorar la eficiencia del sistema. A continuación, se explica brevemente de que se encarga cada función definida en esta interface:

- ***invalidCommandMessage***: esta función se encarga de imprimir un mensaje de "Comando no válido" cuando se recibe un comando que no coincide con ninguno de los comandos esperados (Ver ilustración-1)



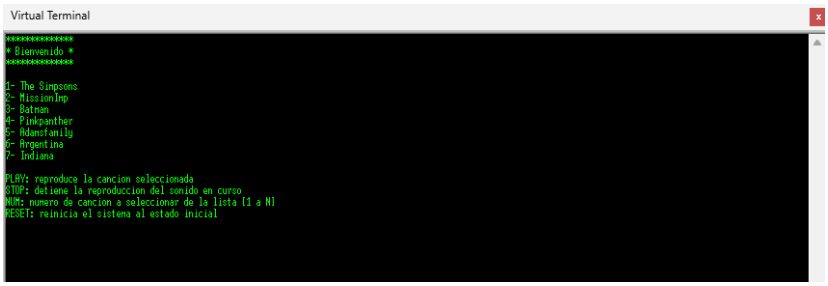
Ilustracion-1 comando no valido

- ***imprimirCanciones***: se encarga de imprimir la lista de canciones disponibles en el sistema. Recorre una estructura de datos que almacena la información de cada canción y formatea los datos para su impresión (ver ilustración-2)



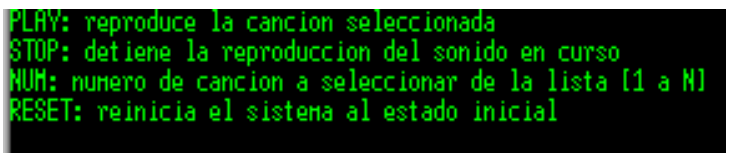
Ilustracion-2 ringtones

- ***welcomeMessage***: muestra un mensaje de bienvenida al sistema. Imprime un encabezado a modo decorativo, el mensaje de bienvenida y un separador. Luego, invoca la función *imprimirCanciones* para mostrar la lista de canciones disponibles. (ver ilustración-3)



Ilustracion-3 bienvenida

- ***optionsMessage***: esta función muestra los diferentes comandos y opciones disponibles en el sistema. Imprime una serie de mensajes que explican cómo usar los comandos disponibles. (ver ilustración-4)



Ilustracion-4 opciones

6.3 Memoria FLASH:

Almacenar tal cantidad de cadena de caracteres para la librería RTTL, la memoria volátil no es suficiente para poder cargar todos los datos necesarios del proyecto. En estos casos se utilizar la memoria no volátil para almacenar datos de forma permanente. Esta también es de solo lectura. Se decidió así, almacenar en memoria FLASH los mensajes que se le otorgan al usuario y que, para poder obtener esos mensajes de memoria no volátil, un buffer lo suficientemente grande para almacenar de manera temporal y así, transmitirlo.

En C, es necesario incluir la biblioteca de AVR que nos provee la funcionalidad de trabajar con este tipo de almacenamiento.

```
#include <avr/pgmspace.h>
```

Siendo la sintaxis para almacenar un string:

```
const char message1[] PROGMEM = "PLAY: reproduce la cancion seleccionada \r\n";
const char message2[] PROGMEM = "STOP: detiene la reproduccion del sonido en
curso \r\n";
const char message3[] PROGMEM = "NUM: numero de cancion a seleccionar de la lista
[1 a N] \r\n";
const char message4[] PROGMEM = "RESET: reinicia el sistema al estado inicial
\r\n";
const char message5[] PROGMEM = "Comando no valido \r\n";
const char message6[] PROGMEM = "* Bienvenido * \r\n";
```

Y para leerlo se utiliza una function de la libreria de Strings de C. Por ejemplo:

```
strcpy(buffer, message5);
```

7. Máquina de estados finitos.

Al igual que en el trabajo desarrollado anteriormente, se vuelve a hacer uso de este modelo, en este caso, para controlar el comportamiento del sistema de reproducción de música, ya que nos permite organizar y gestionar las distintas etapas del proceso.

7.1 Introducción.

Se trabaja con una MEF para controlar la reproducción de canciones, respondiendo a comandos recibidos a través de la comunicación UART y cambiando de estado según las condiciones establecidas.
Aquí es fundamental el driver realizado de la UART, ya que nos permite conocer que comando fue el que se recibió y así, gestionar las transiciones dependiendo del comando. Las funcionas del driver utilizadas son:

`char* UART_Get_String_From_Buffer()` Necesaria para poder obtener el comando escrito por el usuario

`uint8_t UART_Get_Flag()` flag que indica que un comando fue escrito y que el buffer tiene un dato correcto.

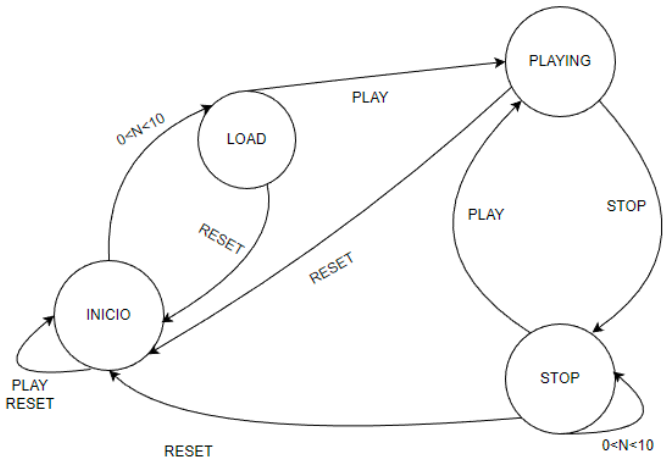
7.2 Implementación.

La MEF se basa en la utilización de una variable que representa el estado actual del sistema, y se emplea una enumeración para distinguir cada estado de forma clara. Para cada estado, se implementa una función específica que gestiona este mismo, con la lógica necesaria para realizar las acciones que corresponden al estado actual, y las transiciones a otros estados.
Para este caso, la máquina de estados finita es de tipo Moore, es decir, la salida solo depende del estado.

7.2.1 Estados:

- **INICIO**: representa el estado inicial del sistema. En este estado, se muestra un mensaje de bienvenida y se ofrecen opciones al usuario.
- **LOAD**: indica que se ha seleccionado una canción para cargar y reproducir. En este estado, se carga la canción seleccionada y se realiza el procesamiento necesario para preparar la reproducción.
- **PLAYING**: indica que la reproducción de la canción está en curso. En este estado, se reproducen las notas de la canción
- **STOP**: representa el estado de pausa de la reproducción. En este, la reproducción se detiene temporalmente, pero se mantiene la configuración de la canción cargada.

En cada uno de los estados se verifica si hay comandos nuevos recibidos (UART_Get_Flag). Obteniendo que comando fue (UART_Get_String_From_Buffer) para realizar cambios en el estado o en el comportamiento de la reproducción. (ver ilustración-5)



Ilustracion-5 MEF MOORE

7.2.2 Transiciones.

- Transición desde el estado **INICIO**:
  - Si se recibe un comando válido para cargar una canción, la MEF transita al estado LOAD.
  - Si se recibe el comando "RESET", la MEF transita de nuevo al estado INICIO.
  - Si se recibe el comando "STOP" y la MEF se encuentra en los estados STOP, INICIO o LOAD, no se produce ninguna transición.
  - Si se recibe cualquier otro comando, se muestra un mensaje de comando inválido y no se produce ninguna transición.
- Transición desde el estado **LOAD**:
  - Si se recibe el comando "PLAY" y la MEF se encuentra en los estados STOP o LOAD, transita al estado PLAYING.
- Transición desde el estado **PLAYING**:
  - Si se recibe un comando válido durante el estado PLAYING, se realiza el manejo correspondiente del comando. Si el comando implica detener la reproducción, la MEF transita al estado INICIO.
  - Si se recibe el comando "STOP" y la MEF se encuentra en el estado PLAYING, transita al estado STOP.
  - Si no se recibe ningún comando y la reproducción de notas ha finalizado, la MEF transita al estado INICIO.
- Transición desde el estado **STOP**:
  - Si se recibe un comando válido durante el estado STOP, se realiza el manejo correspondiente del comando. Si el comando implica reanudar la reproducción, la MEF transita al estado PLAYING.
  - Si no se recibe ningún comando, la MEF permanece en el estado STOP.

8. Conclusión:

Luego del desarrollo del proyecto junto con los nuevos conceptos aplicados como la transmisión serie, específicamente la UART y la arquitectura Background/foreground se puede concluir que son de gran ayuda para la comunicación con el usuario. Es de importancia conocer las maneras de tener una comunicación con el exterior y, en base a la entrada, realizar ciertas decisiones.

Por otro lado, se vio una de las funcionalidades del TIMER1 y poder trabajar con el doble de bits que otros timer. Pudiendo así generar un amplio rango de frecuencias que, en este caso, se utilizó para generar sonidos.

Por último, se pudo apreciar la importancia de almacenar ciertos datos en memoria flash y así, liberar la RAM. Teniendo en cuenta que es escasa y no ilimitada.