



Trabajo Practico N3

20/06/2017

Matías Pan - 783/9

Agustín Sánchez - 939/2

Jorge Stranieri - 917/5

Planteo general

Se realizarán un conjunto de ejercicios con el fin de entender la programación de periféricos externos con puertos de entrada/salida. Para esto utilizamos el kit dado por la cátedra.

Inciso N°5

Consigna

Utilizando los periféricos SCL y TPM, se desea implementar un generador de sonidos controlado con la PC. El generador debe funcionar en base a los diferentes comandos recibidos desde la PC y además de reproducir los sonidos seleccionados, deberá actualizar la información en la pantalla enviando los mensajes adecuados y generando una interfaz amigable al usuario.

- El sistema debe iniciarse con el sonido apagado y con un mensaje de bienvenida en la pantalla de la PC (por única vez) indicando el rango de frecuencias posibles y la lista comandos disponibles para controlar la generación. El rango de frecuencias posible será de 200Hz a 10KHz en pasos de 100Hz.*
- El generador tendrá dos modos de funcionamiento: generación de una señal digital de frecuencia fija o generar un barrido del rango completo de frecuencias. En el primer caso, el usuario deberá elegir la frecuencia del sonido, por ejemplo $f=1500\text{Hz}$. Además, se deberá informar al usuario el error cometido al generar la señal de la frecuencia seleccionada, por ejemplo $\text{Error} < xx \text{ Hz}$. En el segundo caso, el usuario deberá ingresar al modo barrido y seleccionar el tiempo de barrido dentro de tres posibilidades $T1=5\text{seg}$, $T2=10\text{seg}$ y $T3=15\text{seg}$.*
- En ambos modos, el usuario podrá encender o apagar el sonido mediante comandos ON-OFF y en cualquier caso el usuario puede volver al estado inicial con un comando de RESET.*
- La comunicación con la PC mediante el periférico serie del uC, se deberá implementar utilizando Buffers e interrupciones de Transmisión y Recepción. En la salida del canal 1 del TPM1 (PTCD1) se deberá conectar el parlante.*

Interpretación

Se debe implementar un generador de sonidos controlado por la PC. Para esto se conecta un parlante en el puerto C del microcontrolador (PTCD1), mientras que la comunicación del microcontrolador y la PC va a ser mediante una conexión serial utilizando un cable RS-232 conectado al puerto B del microcontrolador. Por este cable le llegarán los comandos al microcontrolador para saber qué función debe cumplir. Además se necesita programar una interfaz amigable para el usuario para que pueda ingresar los comandos correctamente y que el microcontrolador los capte sin ningún

problema. La captación de dichos comandos se realizará mediante interrupciones de los buffers de transmisión y recepción.

Para el ingreso y envío de comandos se usará la HyperTerminal, de esta forma, aseguramos que cualquier carácter que ingresemos será enviado por el cable RS-232.

Los comandos que se presentarán en la hyperterminal para el manejo del generador de sonidos son:

- on
- off
- reset
- f freq (Con freq = Frecuencia que se quiere establecer)
- sweep 5
- sweep 10
- sweep 15

Siendo *sweep* el comando para activar el barrido y luego se le pasa un argumento el cual representa el tiempo de barrido que se desea. Este puede ser 5, 10 o 15, pasar un número diferente a alguno de esos resulta en un error de comando no reconocido.

Resolución

Para la resolución de este trabajo nos enfocamos en tres partes importantes. Por un lado hacer un interprete de comandos o shell, el cual es el medio de comunicación entre el usuario y el sistema. Por otro lado, hacer un modulo independiente que se encargue de la configuración y reproducción de los sonidos. Y finalmente, en implementar dos buffers que manejen la entrada y salida del microcontrolador.

El programa cuenta con las librerías:

- **Sound.c** - Encargada de generar las frecuencias necesarias para la reproducción del sonido.
- **Bufferrx.c** - Encargada de manejar los datos de entrada enviados por el usuario mediante la terminal.
- **Buffertx.c** - Encargada de enviarle mensajes al usuario y mostrar en pantalla lo que se escribe.
- **Shell.c** - Es el corazón del sistema. Se encarga de generar de la comunicación con el usuario e interpretar los comandos recibidos por el mismo.

También están presentes **main.c** y **MCUnit.c** las cuáles tienen el orden de ejecución del programa y la respuesta ante interrupciones respectivamente.

Para efectuar el correcto funcionamiento del microcontrolador usamos y configuramos tres módulos del mismo:

El módulo **SCI** (Serial Communications Interface Module). Este modulo nos facilita la obtención de los caracteres mediante el cable RS-232, ya que nos provee de

interrupciones cada vez que se escribe en la terminal o se tiene que imprimir en pantalla.

Módulo **TMP1** (Timer Interface). Este modulo nos ayuda con la generación de los sonidos a emitir por el parlante, debido a que permite utilizar un clock de alta frecuencia. Con este módulo generamos señales digitales por medio de la función Output Compare.

Módulo **RTC** (Real-Time Counter Module). Utilizamos este modulo para que el barrido de frecuencias se pueda llevar a cabo en las tres posibles opciones de tiempo preestablecidos. Decidimos utilizarlo para el barrido dado que la frecuencia de barrido era lo suficientemente pequeña para representarla con RTC.

Configuraciones de los módulos e interrupciones

Interrupción del módulo SCI

Con este módulo y con la implementación de los buffers de entrada y salida manejamos la lectura y escritura de caracteres respectivamente.

Configuración: Dentro de este módulo primero definimos el baud rate, que es la velocidad de transmisión que seteamos para la comunicación de los caracteres. En este caso seteamos un baud rate divisor de 52, lo que aproximadamente nos deja un baud rate de 9600. Éste mismo baud rate es el que debemos configurar en la HyperTerminal para que funcione correctamente la transmisión.

Luego en las interrupciones de este módulo sólo habilitamos Rx Interrupt. Esto es debido a que cada vez que el usuario ingrese un caracter en la HyperTerminal, el flujo del programa pasa a atender la interrupción de recepción del carácter, guardandolo en el buffer de entrada que hemos implementado, para luego procesar lo ingresado.

Por lo tanto las configuraciones realizadas son:

- Baud rate divisor → 52
- Pin de recepción → PTB0 (por defecto)
- Pin de transmisión → PTB1 (por defecto)
- Transmit request interrupt → disable. Se setea temporalmente en enable por el buffertx cada vez que tiene datos para enviar.
- Receive request interrupt → enable
- Initialization transmitter → enable
- Initialization receiver → enable

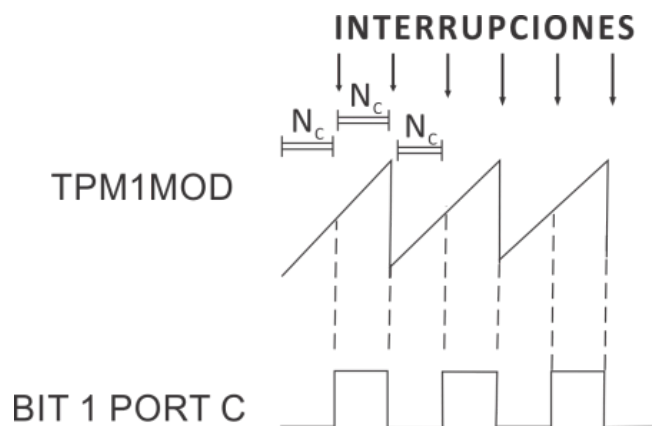
Interrupción módulo TMP1

Nuestro objetivo es utilizar el TPM1 para generar una señal digital cuadrada con un ciclo de trabajo de 50%, es decir mismo tiempo en valor alto que en valor bajo.

Para ello utilizamos la función del TPM1 de Output Compare, en particular con la acción de Toggle la cual conmuta el valor del pin asociado (BIT 1 del PORTC) cada vez que el valor del contador alcanza el valor del registro del canal correspondiente. Como utilizamos el canal 1, dicho registro será **TPMC1V**.

Esto además de realizar un toggle en el pin, dispara una interrupción en la que deberíamos incrementar **TPMC1V** en **Nc** unidades, de manera de asegurar constantemente la generación de la señal cuadrada. Dicha interrupción veremos más adelante que es atendida por la librería **Sound**.

Se puede ver en el siguiente gráfico el funcionamiento del TPM:



La frecuencia de la señal cuadrada generada dependerá del valor **Nc**. Mientras más pequeño N_c mayor la frecuencia, y viceversa. La relación entre la frecuencia deseada de la señal, el valor N_c y la frecuencia del clock del TPM viene dada por:

$$F_{deseada} = \frac{F_{tpmclock}}{2 * N_c}$$

Por lo tanto las configuraciones realizadas son:

- **Clock source** → bus rate clock de 8 MHz
- **Canales** → 1
- **Channel 1 interrupt enable** → se habilita solamente cuando se quiere reproducir el sonido
- **Channel 1 Value** → se ira actualizando a medida que se deseen diferentes frecuencias

Interrupción módulo RTC

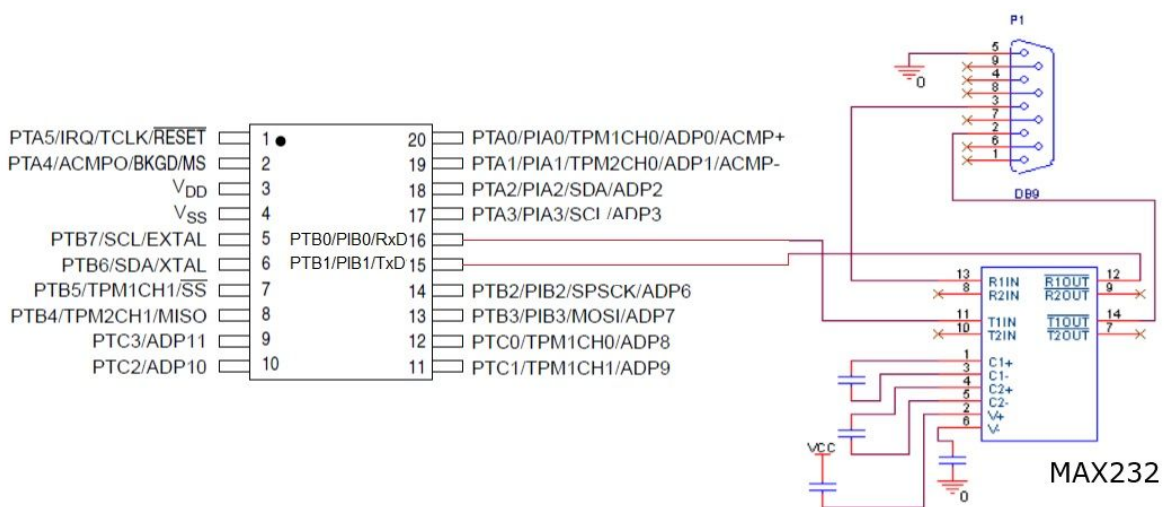
La configuración para el módulo RTC es la más sencilla. Solamente hay que poner en el Prescaler el valor 1, dejando RTC modulo value a disposición de la librería **Sound** para configurar la frecuencia de barrido deseada. Es importante aclarar que el campo RTC

Interrupt debe estar en Disabled. De esta forma comienza el programa con la interrupción deshabilitada, pero en el único momento que se habilita es cuando se llega el comando "sweep" más el argumento de tiempo que se quiera que se reproduzca el barrido. De esta forma, la interrupción se habilita cuando se va a hacer el barrido y es la que controla que se ejecute en el tiempo pedido.

Por lo tanto las configuraciones realizadas son:

- **Clock** → oscilador interno de 1kHz
- **Prescaler** → 1
- **RTC module value** → dependiera del barrido
- **RTC interrupt** → disable. Solo se habilita cuando se realiza el barrido.

Esquema de conexiones(conexión serial):




Hardware y periféricos:

Transductor electroacústico: dispositivo que consta de un electroimán y una lámina metálica de acero. Cuando se acciona, la corriente pasa por la bobina del electroimán y produce un campo magnético variable que hace vibrar la lámina de acero sobre la armadura. Esta vibración viene ligada a la frecuencia de la señal de entrada.

Cable RS-232 (Recommended Standard 232): El RS-232 consiste en un conector tipo DB-9 (de 9 pines). Cable de comunicación serie asíncrono. La configuración establecida para la interfaz con este medio de comunicación es:

- Data Format: 8 bits (Puede transmitir 9).
- Sin paridad.
- 9600 de baud rate.
- Control de flujo: ninguno

Interfaz del usuario:



```

SWTPC Com1 - HyperTerminal
File Edit View Call Transfer Help

Lista de comandos:
-----
ON      >> prender el sonido
OFF     >> apagar el sonido
RESET  >> resetear
SWEEP 5 >> barrido de 5seg
SWEEP 10 >> barrido de 10seg
SWEEP 15 >> barrido de 15seg
F FREQ >> frecuencia fija entre 200-10000 con pasos de 100 Hz
> on
Prendiendo sonido...
> f 1000
Seteando frecuencia 1000
Con error de 0Hz
>

Connected 0:00:41  Auto detect  1200 8-N-1  SCROLL  CAPS  NUM  Capture  Print echo

```

Implementación de los buffers

Usamos un modelo Productor/Consumidor para los buffers de entrada y salida. Están implementados de forma como una pila. El buffer para transmitir tiene una capacidad de 32 caracteres. El buffer de entrada en cambio, recibe un carácter por vez y esos caracteres se van guardando en un arreglo de 16 caracteres de longitud dentro de la librería **shell**. De esta forma, si se ingresan por teclado más de 16 caracteres, se genera un error; el puntero del arreglo vuelve al inicio y se carga en el buffer transmisor un string informando al usuario que se ha excedido la capacidad del buffer.

Generación de sonidos:

Para la generación de los sonidos tenemos que respetar el rango de las frecuencias que se pueden reproducir que son: de 200 Hz a 10000 Hz en pasos de 100 Hz. Esto quiere decir que solo se permiten frecuencias correspondientes a múltiplos de 100, siempre y cuando se mantengan entre ese rango.

Con la ayuda del módulo TPM1 con la función Output Compare facilitamos la generación de una señal de onda cuadrada. Así nos basamos en esta señal para generar la frecuencia deseada de escuchar a través del parlante. Más adelante se verá

las fórmulas necesarias para el cálculo y configuraciones necesarias para cierta frecuencia deseada.

Explicación detallada de cada módulo

Bufferrx

El bufferrx, un buffer de recepción, se encarga de manejar la entrada de los datos al sistema.

Cada vez que se genera un interrupción de recepción en el microcontrolador, se delega su atención a este buffer, el cual almacena el valor de entrada contenido en el registro **SCID** y habilita un flag (**FLAG_RECEIVED**) indicando que se recibió un dato que debería ser consumido por la **shell**.

Por características de la comunicación serial, los datos recibidos serán siempre de 8 bits. Y dado que nos interesa que el intérprete de comandos vaya consumiendo dichos datos a medida que van entrando, nos pareció acorde y suficiente que el almacenamiento sea de un solo campo denominado **car** cuyo tipo es "char".

La interrupción de recepción está habilitada desde un comienzo y en ningún momento se deshabilita, ya que es de nuestro interés estar continuamente escuchando lo que el usuario quiere decir, es decir, lo que el usuario transmite.

Buffertx

De forma análoga se puede definir el buffertx, un buffer de transmisión, que se encarga de enviar los datos hacia la interfaz de usuario (a la HyperTerminal). De todas formas, su funcionamiento lleva una lógica más compleja.

El propósito de este buffer es permitirle a la **shell** transmitir datos (strings y caracteres) al usuario para su interfaz correspondiente. La transmisión no se hace de forma instantánea; los datos se van colocando en el buffer y se van a ir transmitiendo de carácter en carácter a medida que vayan surgiendo las interrupciones de transmisión.

En primer lugar, el tamaño de almacenamiento del buffer es de 32 bytes, es decir, 32 caracteres. Este se comporta de manera circular, por lo que contamos con dos índices:

- **w**: siendo el índice que corresponde a la posición del buffer en que se deberán escribir los datos nuevos.
- **r**: siendo el índice que representa el comienzo de los datos a transmitir.

Ambos índices son **unsigned char**, por lo que tienen una representación de 256.

Para lograr que el buffer sea circular, utilizamos los índices descritos previamente con los módulos necesarios. En este caso, la longitud de nuestro buffer es de 32 caracteres, entonces para asegurar que nuestros índices no se vayan de rango, cada vez que accedemos al valor de ellos hacemos un ``mod 32`` con el fin de obtener la posición del buffer deseada.

A **w** y **r** nunca le hacemos un chequeo para resetearlas en caso de que se vayan de rango, ya que como son variables sin signo lo hacen solas. Para entrar más en detalle, cuando **w** o **r** estén a 255 y quieran avanzar un índice, su valor pasa a ser cero ya que se excede de su máxima representación. Vemos que esto no genera conflictos e incluso nos ayuda a lograr el buffer circular deseado, debido a que si se encuentra con el valor 255, correspondiente a la última posición del buffer($255\%32=31$), el siguiente valor pasa a ser 0, el cual corresponde a la primera posición.

A su vez tenemos una variable interna **empty** la cual indica si el buffer está vacío o si tiene datos que todavía debe transmitir. Dicha variable se inicializa en "true"; cada vez que se ingresa un dato se indica que el buffer no está vacío seteando dicha variable a "false". Esta solo volverá a su estado de "true" cuando la transmisión se haya completado, es decir cuando los índices **w** y **r** sean iguales.

Esta variable es de suma importancia para cuando se intenta escribir reiterados datos en el buffer, lo que podría generar que no tenga el tiempo suficiente para transmitir el dato anterior. Por lo que solo se permite ingresos de nuevos strings cuando **empty==true**, indicativo de que se terminó de transmitir los datos anteriores.

Desde un comienzo, las interrupciones de transmisión están apagadas ya que no tenemos dato que haya que transmitir en el buffer. Luego, cada vez que se agrega un dato a dicho buffer, habilitamos la interrupción de transmisión mediante el registro **SCIC2_TIE**. Estos datos se encuentran disponibles para consumir en el registro **SCID**, una vez que sean leídos completamente, se apaga nuevamente la interrupción de transmisión, dado que el buffer se encuentra vacío.

Sound

La librería sound es la encargada de generar la onda cuadrada de frecuencia variable para la generación del sonido indicado.

Esta librería consta de un set de funciones para permitirle a la **Shell** controlar el sonido. Estas son:

- **sound_on:** habilita la interrupción correspondiente a la generación de sonido.
- **sound_off:** deshabilita el sonido. Si cuando se deshabilito el sonido se estaba corriendo un barrido, éste continúa sin la reproducción de sonido de todas formas.
- **sound_reset:** apaga tanto el sonido como los barridos y configura la frecuencia por defecto.
- **sound_sweep:** recibe como parámetro el tiempo en segundos que debería tardar el barrido entero de 200 a 10000 Hz. Habilita el RTC, setea el modulo para cumplir con el periodo deseado y habilita la interrupción para que **sound_sweep_interrupt** maneje el progreso del barrido.
- **sound_set_frecuency:** establece y valida la frecuencia fija que se le pase por parámetro. Devuelve el error en la representación de la frecuencia, que se

genera debido al redondeo de variables enteras. Deshabilitar si es que se estaba realizando un barrido posteriormente.

También cuenta con funciones para atender las interrupciones:

- **sound_interrupt:** es la encargada de ir actualizando el valor del registro del canal 1 del TPM. **TPMC1V+=nc**, donde **nc** es la cantidad de ciclos de reloj que deben pasar para que se invierta la señal, formando así la onda cuadrada con la frecuencia deseada.
- **sound_sweep_interrupt:** es la encargada de ir aumentando la frecuencia en pasos de 100 Hz, cada vez que se genere una interrupción de RTC con periodo configurado previamente por la función que habilita el sweep. Si se llega a la frecuencia máxima 10000 Hz, se vuelve a la mínima 200 Hz.

Función interna importante:

- **sound_set_nc:** es la que verdaderamente “configura la frecuencia”. Internamente cada vez que la librería quiera cambiar la frecuencia se llama a esta función.

sound_sweep en mas detalle

En primer instancia se calcula la cantidad de pasos que nos llevaría realizar un barrido desde 200 Hz a 10000 Hz con pasos de 100 Hz. Para ello simplemente calculamos:

$$\text{cantidad de pasos de barrido} = \frac{F_{max} - F_{min}}{F_{paso}}$$

$$\text{cantidad de pasos de barrido} = \frac{10000 - 200}{100} = 98$$

Este número de pasos va ser estático sin importar el tiempo total del barrido por lo que se calcula una sola vez. Luego hay que calcular el periodo correspondiente a cada paso del barrido dado cierto argumento. Para ello:

$$\text{periodo de un paso} = \frac{\text{tiempo total del barrido en ms}}{\text{cant pasos}}$$

Finalmente se deberá setear **RTCMOD= (periodo de un paso)-1** para obtener el tiempo total del barrido deseado.

sound_set_nc en mas detalle

El cálculo que se realiza para calcular **nc** teniendo una **Fdeseada** es el siguiente:

$$nc = \frac{F_{clock}}{F_{deseada} \cdot 2}$$

Siendo **Fclock** la frecuencia correspondiente a la del bus de datos → de 8 MHz.

Y el divisor se lo multiplica por dos debido a que la onda cuadrada deseada tiene un ciclo de trabajo de 50%: mismo tiempo en alto que en bajo.

Calculo de error en sound_set_frequency

Debido a que **nc** debe ser un valor entero y que su cálculo puede dar un número decimal, se lo debe redondear al entero más cercano. Si bien este se debe realizar para el debido funcionamiento del TPM, puede generar que la frecuencia verdadera no sea idénticamente igual a la frecuencia deseada → existe un error.

La consigna nos pide que informemos ese error. Para ello se realiza el siguiente cálculo:

$$[nc] = \frac{F_{clock}}{F_{deseada} \cdot 2} \quad (\text{siendo } [] \text{ el valor entero})$$

$$F_{obtenida} = \frac{F_{clock}}{[nc] \cdot 2}$$

$$error = F_{obtenida} - F_{deseada}$$

Main

El main es muy simple y se encarga solamente de la inicialización del sistema y de su funcionamiento. Se lo puede ver en el siguiente código:

```
1. #include "shell.h"
2.
3. void MCU_init(void); /* Device initialization function declaration */
4.
5. void main(void) {
6.     MCU_init();
7.     // shell_reset imprime todos los comandos disponibles y
8.     // resetea las interrupciones
9.     shell_reset();
10.    for (;;) {
11.        // shell_update chequea si hay un comando para ejecutar
12.        shell_update();
13.    }
14. }
```

Shell

La shell es el medio de comunicación entre el usuario y el microcontrolador. Por la HyperTerminal de la computadora se reciben los comandos para las distintas funciones del microcontrolador. Esta librería es la encargada de interpretar y ejecutar esos comandos adecuadamente, llamando a las funciones correspondientes de la librería **sound** para poder realizar la reproducción de las distintas frecuencias.

Debido a que se nos pide realizar una interfaz de usuario amigable, decidimos que la librería shell también se encargaría de simular una consola. Es la parte más importante de nuestro sistema. Desde esta librería mandamos los strings para informar al usuario cuáles son los comandos que se pueden utilizar y de que forma; una vez que se ingresa por teclado algún carácter es guardado dentro de un arreglo para luego poder ser

procesado e indicar cuál es la acción a realizar. Básicamente la mayor parte del flujo del programa pasa por esta librería.

La principal tarea de esta librería es interpretar lo que el usuario ingresa y hacer lo correspondiente a lo ingresado. Para esto utilizamos la función **shell_update**. Esta función es llamada constantemente desde el **main**. Su trabajo es consumir los caracteres enviados por el usuario, los cuales están disponibles en el buffer de recepción, y una vez que se recibe el carácter “\r”, el cual representa un enter, **shell_update** llama a la función encargada de hacer la ejecución de comandos. En caso de que se reciban 16 caracteres sin recibir un enter, esta función le indica al usuario que el comando que está intentando ingresar esta fuera de rango.

Para la ejecución de los comandos, nuestra función principal es **shell_execute**. Esta función actúa como un despachador de comandos. Lo primero que hace es consumir el buffer que contiene los caracteres que se ingresaron. Si los caracteres ingresados corresponden a un comando que nuestro programa reconoce, entonces esta procede a llamar a la función encargada de manejar dicho comando. En caso de que lo que se ingreso no se reconozca, **shell_execute** imprimirá en pantalla un mensaje útil indicando que lo que se ingreso no es un comando que sepamos ejecutar.

Como mencionamos previamente, **shell_execute** es la encargada de llamar a la función correspondiente de cada comando. Estas funciones son:

- **shell_on**: esta función avisa que se va a prender el sonido y llama a la función **sound_on** explicada previamente.
- **shell_off**: avisa que se va a apagar el sonido y llama a la función **sound_off** de la librería sound.
- **shell_reset**: imprime todos los comandos disponibles llamando a **shell_show_commands** y luego llama a la función **sound_reset** la cual resetea todas las configuraciones.
- **shell_sweep_5**: esta función llama a **sound_sweep** pasandole 5 como argumento, el cual indica el tiempo de barrido deseado.
- **shell_sweep_10**: esta función llama a **sound_sweep** pasandole 10 como argumento, el cual indica el tiempo de barrido deseado.
- **shell_sweep_15**: esta función llama a **sound_sweep** pasandole 15 como argumento, el cual indica el tiempo de barrido, en segundos, deseado.
- **shell_num**: esta función recibe por parámetro la frecuencia que se desea setear. Previo a setearla, hace los chequeos necesarios para que la frecuencia pasada sea válida. Una vez que se confirme que es válida, se llama a **sound_set_frequency** la cual setea la frecuencia y devuelve el error. Luego imprimimos el error de dicha frecuencia.

Finalmente, tenemos dos funciones más. **shell_show_commands** la cual simplemente imprime una lista de los comandos que la shell reconoce. Y **shell_error** la cual imprime que el comando ingresado no es reconocido.

Anexo: Archivos de C

Para una visión más detallada del código, consultar el repositorio:

<https://github.com/trorik23/CDyuC-TP3>

Main.c

```

1. #include <hidef.h> /* for EnableInterrupts macro */
2. #include "derivative.h" /* include peripheral declarations */
3. #include "shell.h"
4.
5. #ifdef __cplusplus
6. extern "C"
7. #endif
8. void MCU_init(void); /* Device initialization function declaration */
9.
10. void main(void) {
11.     MCU_init();
12.     // shell_reset imprime todos los comandos disponibles y
13.     // resetea las interrupciones
14.     shell_reset();
15.     for (;;) {
16.         // shell_update chequea si hay un comando para ejecutar
17.         shell_update();
18.     }
19. }
```

Shell.h

```

1. #ifndef SHELL_H_
2. #define SHELL_H_
3.
4. // shell_execute actúa como dispatcher para los comandos. Esta consume el buffer
5. // que contiene los caracteres ingresados (sin incluir el enter), y ejecuta el
6. // comando correspondiente. Las funciones de los comandos están almacenadas en un
7. // arreglo previamente definido.
8. void shell_execute(void);
9.
10. // shell_reset muestra la lista de comandos y resetea todas las configuraciones
11. // Esto hace que no se escuche ningún sonido
12. void shell_reset(void);
13.
14. // shell_update chequea mediante el FLAG_RECEIVED si hay un carácter disponible
15. // en caso de que haya y el carácter sea "\r" (enter) entonces procede a ejecutar
16. // el comando. En caso de que no sea enter, guarda el carácter en el buffertx. Además
17. // chequea que la cantidad de caracteres que se ingresó previo a enter no supere el
18. // rango máximo.
19. void shell_update(void);
20. #endif
```

Shell.c

```

1. #include "shell.h"
2. #include "bufferrx.h"
3. #include "sound.h"
4. #include "buffertx.h"
5.
6. #define LEN 16
7. char buff[LEN];
8. char ibuff = 0;
9.
10. void push_upper_to_lower_case(char);
11.
12. // comando: sweep 5
13. void shell_sweep_5(void);
14. // comando: sweep 10
15. void shell_sweep_10(void);
16. // comando: sweep 15
17. void shell_sweep_15(void);
18. // comando: on
19. void shell_on(void);
20. // comando: off
21. void shell_off(void);
22. // comando: f 300
23. void shell_num(unsigned int);
24.
25. void shell_error(void);
26. void shell_show_commands(void);
27.
28. const char NUMBER_OF_COMMANDS = 6;
29. const char *COMMANDS[] = { "on", "off", "reset", "sweep 5", "sweep 10",
30.     "sweep 15" };
31. const char COMMANDS_LEN[] = { 2, 3, 5, 7, 8, 8 };
32. void (*COMMANDS_FUNC[])(void) = { shell_on, shell_off, shell_reset,
33.     shell_sweep_5, shell_sweep_10, shell_sweep_15 };
34.
35. // devuelve 1 si str=command dentro de buff
36. // 0 caso contrario
37. char shell_compare(char * str, char length) {
38.     char aux = 0;
39.     // considerando los strings con '\0' final
40.     while (aux < length && str[aux] != '\0' && buff[aux] == str[aux])
41.         aux++;
42.     if (aux == length && str[aux] == '\0')
43.         return 1;
44.     return 0;
45. }
46.
47. // shell_execute actua como dispatcher para los comandos. Esta consume el buffer
48. // que contiene los caracteres ingresados(sin incluir el enter), y ejecuta el
49. // comando correspondiente. Las funciones de los comandos estan almacenadas en un
50. // arreglo previamente definido.
51. void shell_execute(void) {
52.     char r = 0;

```

```

53.     unsigned int num = 0;
54.     if (shell_compare("f ", 2)) {
55.         // si el primer caracter es f, entonces se llama al comando
56.         // de frecuencia
57.         r = 2;
58.         while (r < ibuff && buff[r] - '0' >= 0 && buff[r] - '0' <= 9) {
59.             // es un digito
60.             num=num*10;
61.             num+=buff[r]-'0';
62.             r++;
63.         }
64.         if (r == ibuff) {
65.             // todos digitos, entonces llamamos al comando
66.             shell_num(num);
67.             return;
68.         }
69.         // si llegamos aca algo salio mal, por ejemplo no es un numero lo
70.         // que se paso, entonces mostramos error
71.         shell_error();
72.         return;
73.     }
74.     // si no corresponde a la instruccion de setear frecuencia sigue
75.     // COMMANDS_FUNC es un array de funciones de todos los comandos que quedan, sin
76.     // incluir el comando de setear frecuencia
77.     for (r = 0; r < NUMBER_OF_COMMANDS; r++) {
78.         if(COMMANDS_LEN[r]!=ibuff) continue;
79.         if(shell_compare(COMMANDS[r],ibuff)) {
80.             (*COMMANDS_FUNC[r])();
81.             return;
82.         }
83.     }
84.     // si llegamos aca quiere decir que no reconocemos el comando
85.     shell_error();
86. }
87.
88. // shell_update chequea mediante el FLAG_RECEIVED si hay un caracter disponible
89. // en caso de que haya y el caracter sea "\r"(enter) entonces procede a ejecutar
90. // el comando. En caso de que no sea enter, guarda el caracter en el buffertx. Ademas
91. // chequea que la cantidad de caracteres que se ingreso previo a enter no supere el
92. // rango maximo.
93. void shell_update(void) {
94.     char car;
95.     // FLAG_RECEIVED se setea a uno dentro de la interrupcion de recepcion
96.     if (FLAG_RECEIVED == 0)
97.         return;
98.     car = bufferrx_get_char();
99.     FLAG_RECEIVED = 0;
100.    if (car != '\r')
101.        buffertx_send_char(car);
102.    if (car == '\r') {
103.        if (ibuff > 0) {
104.            shell_execute(); // se llama a la funcion que ejecuta el comando
105.        }
106.        ibuff = 0;

```

```

107.         return;
108.     }
109.     push_upper_to_lower_case(car);
110.     if (ibuff == LEN) {
111.         ibuff = 0;
112.         buffertx_send_str("\r\nCOMANDO FUERA DE RANGO");
113.         buffertx_send_str("\r\n > ");
114.     }
115. }
116.
117. // shell_sweep_5 realiza un barrido de 5 segundos de duracion. La logica del
    barrido
118. // esta dentro de la funcion sound_sweep definida en la libreria sound
119. // > sweep 5
120. void shell_sweep_5(void) {
121.     buffertx_send_str("\r\nBarriendo con T1 = 5s");
122.     buffertx_send_str("\r\n > ");
123.     sound_sweep(5);
124. }
125.
126. // shell_sweep_10 realiza un barrido de 10 segundos de duracion. La logica del
    barrido
127. // esta dentro de la funcion sound_sweep definida en la libreria sound
128. // > sweep 10
129. void shell_sweep_10(void) {
130.     buffertx_send_str("\r\nBarriendo con T2 = 10s");
131.     buffertx_send_str("\r\n > ");
132.     sound_sweep(10);
133. }
134.
135. // shell_sweep_15 realiza un barrido de 15 segundos de duracion. La logica del
    barrido
136. // esta dentro de la funcion sound_sweep definida en la libreria sound
137. // > sweep 15
138. void shell_sweep_15(void) {
139.     buffertx_send_str("\r\nBarriendo con T3 = 15s");
140.     buffertx_send_str("\r\n > ");
141.     sound_sweep(15);
142. }
143.
144. // shell_on habilita el sonido. Va a sonar lo que se habia ejecutado previamente,
145. // es decir, si se hizo un `sweep 5` y luego un `on` se va a escuchar el barrido
146. // de 5 segundos
147. // > on
148. void shell_on(void) {
149.     buffertx_send_str("\r\nPrendiendo sonido...");
150.     buffertx_send_str("\r\n > ");
151.     sound_on();
152. }
153.
154. // shell_off apaga el sonido, pero los barridos o el sonido de frecuencia sigue
155. // corriendo, simplemente no se escucha
156. // > off
157. void shell_off(void) {

```



```

158.     buffertx_send_str("\r\nApagando sonido...");
159.     buffertx_send_str("\r\n > ");
160.     sound_off();
161. }
162.
163. // shell_reset muestra la lista de comandos y resetea todos las configuraciones
164. // Esto hace que no se escuche ningun sonido
165. // > reset
166. void shell_reset(void) {
167.     shell_show_commands();
168.     sound_reset();
169. }
170.
171. // shell_show_commands muestra la lista de comandos disponibles
172. void shell_show_commands(void) {
173.     buffertx_send_str("\r\nLista de comandos:");
174.     buffertx_send_str("\r\n-----");
175.     buffertx_send_str("\r\nON      >> prender el sonido");
176.     buffertx_send_str("\r\nOFF     >> apagar el sonido");
177.     buffertx_send_str("\r\nRESET   >> resetear");
178.     buffertx_send_str("\r\nSWEEP 5  >> barrido de 5seg");
179.     buffertx_send_str("\r\nSWEEP 10 >> barrido de 10seg");
180.     buffertx_send_str("\r\nSWEEP 15 >> barrido de 15seg");
181.     buffertx_send_str("\r\nF FREQ   >>");
182.     buffertx_send_str(" frecuencia fija entre 200-10000");
183.     buffertx_send_str(" con pasos de 100 Hz");
184.     buffertx_send_str("\r\n > ");
185. }
186.
187. // shell_error simplemente imprime "Comando no reconocido" y el caracter
188. // de terminal
189. void shell_error(void) {
190.     buffertx_send_str("\r\nComando no reconocido");
191.     buffertx_send_str("\r\n > ");
192. }
193.
194. // shell_num recibe por parametro un numero que indica la frecuencia que
195. // se desea setear. Ademas realiza los chequeos necesarios para que la
196. // frecuencia provista sea solo de pasos de 100 entre 200 y 10000 Hz.
197. // > f 500
198. // > f 540 => error
199. void shell_num(unsigned int num) {
200.     char error, j, i;
201.     char aux[3];
202.     if ((num < MIN) || (num > MAX)) {
203.         buffertx_send_str("\r\nSolo entre 200-10000 Hz");
204.         buffertx_send_str("\r\n > ");
205.         return;
206.     }
207.     if (num % STEP_FREQ != 0) {
208.         buffertx_send_str("\r\nSolo en pasos de 100 Hz");
209.         buffertx_send_str("\r\n > ");
210.         return;
211.     }

```

```

212.
213.     buffertx_send_str("\r\nSeteando frecuencia ");
214.     // imprimo frecuencia ingresada
215.     for (j = 2; j < ibuff; j++)
216.         buffertx_send_char(buff[j]);
217.     error = sound_set_frequency(num);
218.     // imprimo el error
219.     buffertx_send_str("\r\nCon error de ");
220.     // convertir a string
221.     if (error < 0) {
222.         buffertx_send_char('-');
223.         error *= -1;
224.     }
225.     if (error == 0)
226.         buffertx_send_char('0');
227.     else {
228.         j = 0;
229.         while (error != 0) {
230.             aux[j++] = error % 10;
231.             error /= 10;
232.         }
233.         for (i = j; i > 0; i--)
234.             buffertx_send_char(aux[i - 1] + '0');
235.     }
236.     buffertx_send_str("Hz");
237.     buffertx_send_str("\r\n > ");
238. }
239.
240. void push_upper_to_lower_case(char c) {
241.     if (c >= 'A' && c <= 'Z') {
242.         // mayusculas a minusculas
243.         c = c + ('a' - 'A');
244.     }
245.     buff[ibuff++] = c;
246. }

```

Sound.h

```

1. #ifndef SOUND_H_
2. #define SOUND_H_
3.
4. #define MIN 200
5. #define MAX 10000
6. #define STEP_FREQ 100
7.
8. // sound_interrupt se llama desde la interrupcion de el CH1 del
9. // TPM para setear la frecuencia correspondiente
10. void sound_interrupt(void);
11.
12. // sound_sweep_interrupt se llama dentro de la interrupcion
13. // RTC, esta va realizando en pasos incrementales el barrido
14. // correspondiente.
15. void sound_sweep_interrupt(void);
16.

```

```

17. // sound_set_frequency setea la frecuencia de sonido llamando
18. // a sound_set_nc, y calcula el error de dicha frecuencia.
19. // Recibe la frecuencia que se quiere setear y devuelve el error
20. // correspondiente.
21. char sound_set_frequency(unsigned int);
22.
23. // sound_on prende el sonido mediante la habilitacion de la
24. // interrupcion. Este se llama desde `shell_on`
25. void sound_on(void);
26.
27. // sound_off apaga el sonido mediante la deshabilitacion de la
28. // interrupcion. Este se llama desde `shell_off`
29. void sound_off(void);
30.
31. // sound_reset deshabilita la interrupcion de barrido y apaga el sonido.
32. void sound_reset(void);
33.
34. // sound_sweep realiza un barrido de cierta cantidad de tiempo
35. // en pasos de 100 Hz.
36. // Recibe el periodo del barrido, en el caso de la shell va a ser
37. // 5, 10 o 15. El parametro se toma como segundos
38. void sound_sweep(char);
39.
40. #endif

```

Sound.c

```

1. #include "sound.h"
2. #include "derivative.h"
3.
4. #define STEPS ((MAX-MIN)/STEP_FREQ) // cantidad de pasos para el barrido
5.
6. unsigned int nc = (unsigned int) (4000000 / MIN); // por default
7. unsigned int sweep_freq;
8.
9. // sound_set_nc setea la frecuencia deseada
10. void sound_set_nc(unsigned int freq){
11.     // Fclk=8MHz
12.     // Ciclo de trabajo 50%
13.     // f=frecuencia deseada
14.     // nc= Fclk/(f*2)
15.     nc = (unsigned int) (4000000 / freq);
16. }
17.
18. // sound_on prende el sonido mediante la habilitacion de la
19. // interrupcion. Este se llama desde `shell_on`
20. void sound_on(void) {
21.     // habilitar la interrupcion
22.     TPM1C1SC = 0x14U;
23.     TPM1C1SC_CH1IE = 1;
24. }
25.
26. // sound_off apaga el sonido mediante la deshabilitacion de la

```

```

27. // interrupcion. Este se llama desde `shell_off`
28. void sound_off(void) {
29.     //deshabilitar la interrupcion para que no suene
30.     TPM1C1SC = 0x00U;
31. }
32.
33. // sound_interrupt se llama desde la interrupcion de el CH1 del
34. // TPM para setear la frecuencia correspondiente
35. void sound_interrupt(void) {
36.     TPM1C1V += nc;
37.     TPM1C1SC_CH1F = 0;
38. }
39.
40. // sound_set_frequency setea la frecuencia de sonido llamando
41. // a sound_set_nc, y calcula el error de dicha frecuencia.
42. // Recibe la frecuencia que se quiere setear y devuelve el error
43. // correspondiente.
44. char sound_set_frequency(unsigned int freq) {
45.     RTCSC_RTIE = 0; //deshabilitar sweep
46.     sound_set_nc(freq);
47.     // devolver error, esto se calcula de la siguiente manera:
48.     // f deseada - f obtenida
49.     // f obtenida = Fclk/((nc)*2)
50.     return (4000000 / nc) - freq;
51. }
52.
53. // sound_sweep realiza un barrido de cierta cantidad de tiempo
54. // en pasos de 100 Hz.
55. // Recibe el periodo del barrido, en el caso de la shell va a ser
56. // 5, 10 o 15. El parametro se toma como segundos
57. void sound_sweep(char period) {
58.     unsigned int step_period;
59.     sweep_freq = MIN;
60.     sound_set_nc(sweep_freq);
61.     step_period = (period * 1000) / STEPS;
62.     RTCMOD = step_period - 1;
63.     RTCSC_RTIE = 1; // habilitar interrupcion
64. }
65.
66. // sound_sweep_interrupt se llama dentro de la interrupcion
67. // RTC, esta va realizando en pasos incrementales el barrido
68. // correspondiente.
69. void sound_sweep_interrupt(void) {
70.     sweep_freq += STEP_FREQ;
71.     if (sweep_freq > MAX)
72.         sweep_freq = MIN;
73.     sound_set_nc(sweep_freq);
74.     RTCSC_RTIF = 1; // interrupcion atendida
75. }
76.
77. // sound_reset deshabilita la interrupcion de barrido y apaga el sonido.
78. void sound_reset(void) {
79.     // por default se setea la minima
80.     sound_set_nc(MIN);

```

```

81.    // apagamos el sonido
82.    sound_off();
83.    RTCSC_RTIE = 0; // deshabilitar interrupcion sweep
84.
85. }

```

Bufferrx.h

```

1.  #ifndef BUFFERRX_H_
2.  #define BUFFERRX_H_
3.
4.  // FLAG_RECEIVED indica que se leyo un caracter
5.  extern volatile char FLAG_RECEIVED;
6.
7.  // bufferrx_get_char devuelve el ultimo caracter leido
8.  char bufferrx_get_char(void);
9.
10. // bufferrx_receive_interrupt es llamado desde la interrupcion
11. // de recepcion, un llamado a esta funcion indica que se ingreso
12. // un caracter. Dicho caracter va a estar disponible en el SCID,
13. // el cual se almacena en la variable car. Luego mediante el
14. // FLAG_RECEIVED se le avisa a los que escuchan que se recibio un caracter
15. void bufferrx_receive_interrupt(void);
16.
17. #endif

```

Bufferrx.c

```

1.  #include "bufferrx.h"
2.  #include "derivative.h"
3.
4.  // FLAG_RECEIVED indica que se leyo un caracter
5.  volatile char FLAG_RECEIVED=0;
6.
7.  char car;
8.
9.  // bufferrx_receive_interrupt es llamado desde la interrupcion
10. // de recepcion, un llamado a esta funcion indica que se ingreso
11. // un caracter. Dicho caracter va a estar disponible en el SCID,
12. // el cual se almacena en la variable car. Luego mediante el
13. // FLAG_RECEIVED se le avisa a los que escuchan que se recibio un caracter
14. void bufferrx_receive_interrupt(){
15.     car=SCID;
16.     FLAG_RECEIVED=1;
17. }
18.
19. // bufferrx_get_char devuelve el ultimo caracter leido
20. char bufferrx_get_char(void){
21.     return car;
22. }

```

Buffertx.h

```

1. #ifndef BUFFERTX_H_
2. #define BUFFERTX_H_
3.
4. // buffertx_send_str recibe un string y lo envia caracter por
5. // caracter a la pantalla para que sea impreso.
6. // Recibe str, el cual debe tener el indicador de fin de string
7. void buffertx_send_str(char *);
8.
9. // buffertx_send_char recibe un caracter, lo coloca en el buffer
10. // de impresion y avisa al SCI que hay un caracter que debe ser impreso
11. void buffertx_send_char(char);
12.
13. // buffertx_transmit es llamado desde la interrupcion de transmision
14. // su fin es enviar a la pantalla un caracter
15. void buffertx_transmit(void);
16.
17. #endif

```

Buffertx.c

```

1. #include "derivative.h"
2. #include "buffertx.h"
3. #define LEN 32
4.
5. // buffertx_buff contiene los caracteres que se van transmitiendo
6. // este es un buffer circular
7. char buffertx_buff[LEN];
8.
9. // w indica la posicion en la que se escribio el ultimo caracter
10. // en el buffer
11. // w nunca se resetea, se usan modulos de la longitud del buffer
12. // para no irse de rango
13. unsigned char w;
14.
15. // r indica la posicion en la que se debe leer un caracter, esta es
16. // actualizada desde el buffertx_transmit, es decir, desde la interrupcion
17. // de transmision
18. // r nunca se resetea, se usan modulos de la longitud del buffer
19. // para no irse de rango
20. unsigned char r;
21.
22. // indica que ya se leyó todo lo que había para leer
23. volatile char empty=1;
24.
25. // buffertx_send_str recibe un string y lo envia caracter por
26. // caracter a la pantalla para que sea impreso.
27. // Recibe str, el cual debe tener el indicador de fin de string
28. void buffertx_send_str(char * str) {
29.     while(empty==0);
30.     while (*str != '\0') {
31.         buffertx_send_char(*str);
32.         str++;
33.     }
34. }
35.
36. // buffertx_send_char recibe un caracter, lo coloca en el buffer

```

```

37. // de impresion y avisa al SCI que hay un caracter que debe ser impreso
38. void buffertx_send_char(char c) {
39.     buffertx_buff[(w++) % LEN] = c;
40.     empty=0;
41.     SCIC2_TIE = 1;
42. }
43.
44. // buffertx_transmit es llamado desde la interrupcion de transmision
45. // su fin es enviar a la pantalla un caracter
46. void buffertx_transmit() {
47.     SCID = buffertx_buff[(r++) % LEN];
48.     if(r == w){
49.         empty=1;
50.         SCIC2_TIE=0;
51.     }
52. }

```

MCUInit.c:

El código que se muestra a continuación no es el archivo entero del MCUInit, dejamos únicamente el código de las interrupciones que utilizamos. La configuración de dichas interrupciones fue explicada previamente.

```

1. #include <mc9s08sh8.h> /* I/O map for MC9S08SH8CPJ */
2. #include "MCUinit.h"
3. #include "buffertx.h"
4. #include "bufferrx.h"
5. #include "sound.h"
6.
7. __interrupt void isrVrtc(void)
8. {
9.     sound_sweep_interrupt();
10. }
11.
12. __interrupt void isrVscitx(void)
13. {
14.     // si hay un caracter para transmitir se llama al buffer de transmision
15.     if(SCIS1_TDRE == 1)
16.         buffertx_transmit();
17. }
18.
19. __interrupt void isrVscirx(void)
20. {
21.     // si hay un caracter disponible se llama al buffer de recepcion para que lo
    almacene
22.     if(SCIS1_RDRF == 1)
23.         bufferrx_receive_interrupt();
24. }
25.
26. __interrupt void isrVtpm1ch1(void)
27. {
28.     sound_interrupt();
29. }

```