Circuitos Digitales y Microcontroladores (E0305)
Junio 2020

TRABAJO PRACTICO NRO. 3

Autor:
Nehuen Pereyra (878/6)

1. Interpretación del problema

El objetivo de este trabajo es el desarrollo de software que implementa un generador de sonidos programable de onda cuadrada controlado por comunicación serial basada en comandos. Este programa se ejecutará en el simulador del microcontrolador utilizado por la cátedra (MC9S08SH).

El programa debe poder:

- Aceptar comandos por comunicación serial de recepción y respuesta.
- Se debe poder prender y apagar la reproducción de un sonido.
- Se debe poder reproducir sonidos de onda cuadrada de frecuencia fija (en el rango de 10 Hz a 1000 Hz).

1.1. Detalle del problema a resolver

Para realizar lo anteriormente mencionado disponemos de los siguientes comandos a implementar:

Comando de envío:

STX	CMD-ID	N	DATA	CHKS	ETX
0x02	0xCC	0xNN	...	0xSS	0x03

Figura 1: Comando de envio.

STX: inicio de transmisión.

CMD-ID: es el número de comando, los cuales son 0x01 para encender el generador de sonido, 0x02 para especificar la frecuencia y 0x03 para apagar el generador. En el caso de encender y apagar el generador el campo de DATA será 0x00.

N: indica la cantidad de bytes que contiene el campo DATA.

DATA: indica el valor en Hz de la frecuencia a generar.

CHKS: es el campo verificador del mensaje (1 byte) y es el resultado de la operación XOR de los campos STX, CMD-ID, N y DATA.

ETX: fin de transmisión.

Comando de respuesta:

El sistema responde a cada comando de envió con uno de respuesta, en el cual se calcula el campo CHKS (para verificar de la integridad del mensaje) respondiendo error de checksum si no coincide con el CHKS recibido además en el campo de DATA se envía información sobre si el mensaje enviado pudo ser procesado correctamente.

STX	CMD-ID	N	DATA	CHKS	ETX
0x02	0xCC	0x01	...	0xSS	0x03

Figura 2: Comando de recepción.

Los campos son los mismos que en el envío salvo:

STX: inicio de transmisión

CMD-ID: es el número de comando al cual responde.

N: indica la cantidad de bytes que contiene el campo DATA que en este caso será 0x01.
DATA: puede valer: 0xAA aceptado, 0x55 rechazado, 0xFF error de checksum.
CHKS: es el resultado (1 byte) de la operación XOR de los campos STX, CMD-ID, N y DATA.
ETX: fin de transmisión.

El normal funcionamiento sería el siguiente:

Al iniciar el programa, se encuentra apagado el módulo de sonido, se debe poder a través de los comandos establecer una frecuencia a reproducir y el sistema responder si recibió correctamente el comando, luego se debe enviar un segundo comando para prender la reproducción de sonido y "escuchar" la frecuencia establecida, a esto el sistema responde si recibió correctamente este segundo comando, por último se puede enviar el comando de apagado para silenciar el sonido y el sistema responderá si pudo ejecutar el apagado correctamente.

2. Implementación

Se intentó de modularizar todo el problema para hacerlo más independiente y reutilizable. Se separaron en dos carpetas los distintos módulos que utilizaremos, la primera llamada io de (input/output) donde encontramos:

- **Buffer circular**: El módulo serial utilizara este buffer para la transmisión y recepción de datos (dos buffers independientes).

La segunda carpeta llamada Lib posee los siguientes módulos:

- **Serial**: el cual implementa la función de comunicación serial utilizando el módulo SCI que brinda el microcontrolador.
- **Command**: este módulo se encarga de procesar el comando enviado por el usuario y generar un comando de respuesta.
- **Sound**: este módulo se encarga de todo lo relacionado con el sonido para eso utiliza el módulo TPM.

El archivo main se encarga de inicializar el microcontrolador y los distintos módulos para luego quedar ejecutando las interrupciones que se van generando.

2.1. Serial

Este módulo se encarga de la comunicación serial, donde se utiliza el módulo SCI (Serial Communication Interface) del microcontrolador. El módulo SCI provee un enlace de comunicación UART full-duplex.

Se configuró el módulo SCI para que transmita y reciba a 9600 baudios sin control de flujo ni detección de error por paridad de bits. También se definen los manejadores de interrupciones, esta configuración se puede ver en la figura 3.

Durante la ejecución del programa en el microcontrolador se pueda recibir y/o transmitir caracteres simultáneamente. Para esto, el programa utiliza dos interrupciones disponibles : una interrupción que notifica al programa que el hardware está listo para aceptar un carácter a transmitir (Transmit Data Register Empty) y otra que notifica cuando el hardware ha recibido y colocado un carácter

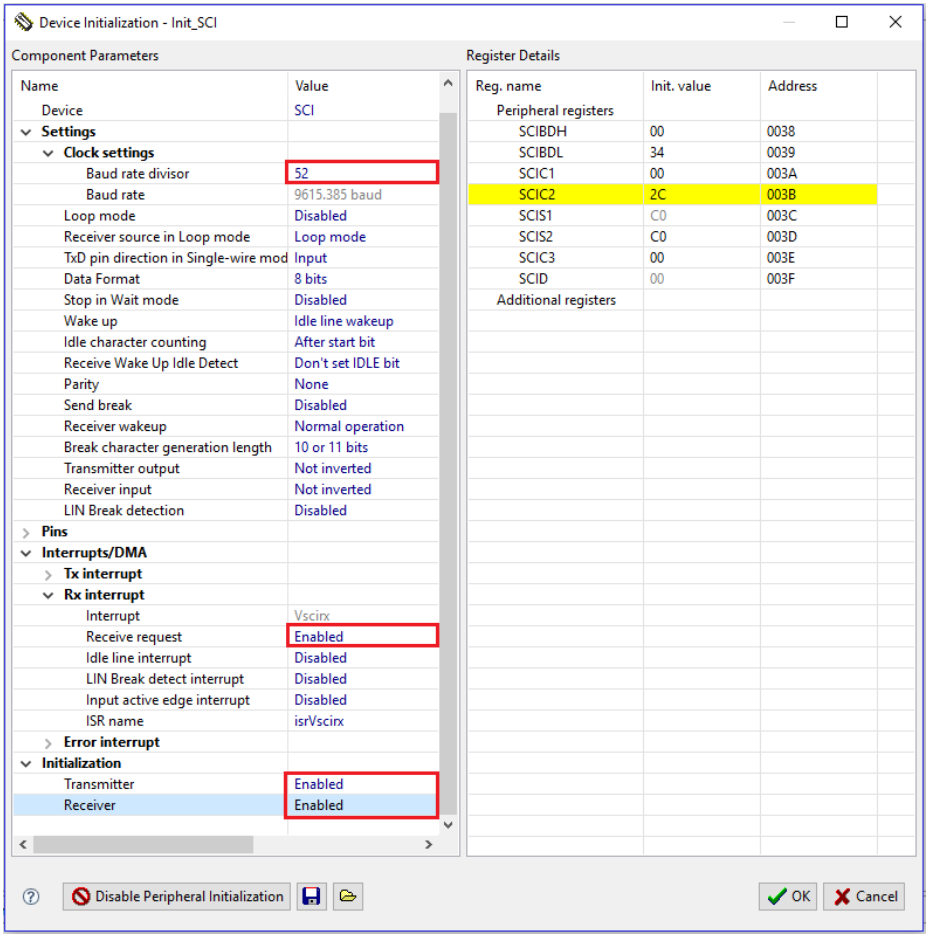

Figura 3: Configuración del modulo SCI.

en el registro de recepción (Receive Data Register Full). Estas interrupciones son **isrVscitx()** e **isrVscirx()** que fueron generadas al momento de configurar el módulo SCI.

Esto implica que el programa debe poder almacenar los caracteres recibidos en cualquier momento, y también debe poder transmitir caracteres en superposición de la ejecución de alguna subrutina. Para esto se necesita buffers en los que se almacenan los caracteres a ser enviados o recibidos de forma independiente (buffer tx y buffer rx).

Se definen dos funciones para la comunicación:

- **serial_send**: Envía una string por serial asincrónicamente, retorna inmediatamente si logra dejar todo el mensaje en el buffer de transmisión buffer tx. Si el buffer estuviera lleno, espera a que se libere lugar en el y almacena el resto de los caracteres.
- **serial_recv**: Recibe una string de 6 o 7 caracteres que representan al comando, retorna una vez ingresado el comando.
- **serial_tx_handler**: Manejador de interrupción interno a ejecutarse cada vez que el hardware puede mandar un caracter por puerto serie.
- **serial_rx_handler**: Manejador de interrución interno a ejecutarse cada vez que el hardware dispone un caracter por puerto serial.

El pseudocódigo de serial_tx_handler es el siguiente:

```
1  Función serial_tx_handler()
2          Si Buffer de recepción esta vacio
3                  Se avisa que se envia datos
4          Sino
5                  Se lee el registro SCIS1_TDRE para el reconocimiento
6                  Se obtiene el dato que se envio
7                  Se guarda en registro SCID
8  Fin serial_tx_handler()
```

Pseudocódigo de la función serial_tx_handler().

El pseudocódigo de serial_rx_handler es el siguiente:

```
1  Función serial_rx_handler()
2          Leer el registro SCIS1_RDRF
3          Agregar el dato en el buffer de recepción
4          Avisar que se recibe los datos
5          Si buffer de recepción esta lleno
6                  Deshabilitar la interrupción, para no recibir datos
7  Fin serial_rx_handler()
```

Pseudocódigo de la función serial_rx_handler().

2.2. Buffers

Los buffers de transmisión y recepción son colas circulares de con una capacidad de 32 caracteres cada una. Estos buffers tienen las siguientes operaciones:

- **buffer_push**: Inserta un carácter en la cola circular, si es que no está lleno.
- **buffer_pop**: Extrae el último carácter ingresado, si es que no está vacío.
- **buffer_empty**: Determina si el buffer está vacío.
- **buffer_full**: Determina si el buffer está lleno.

Los buffers son accedidos por el programa principal durante la ejecución de serial_send o serial_recv y por los manejadores de interrupciones encargados de mandar y recibir caracteres. Esto implica que existe un problema de acceso concurrente a memoria compartida, con lo que hay que tomar medidas para sincronizar el acceso a estas estructuras de datos. Para esto, cada vez que el programa principal va a acceder alguno de los buffers, desactiva las interrupciones correspondientes hasta terminar de acceder a él.

2.3. Command

Este módulo se encarga del procesamiento del comando ingresado por el usuario y la generación del comando de respuesta. Para esto utilizamos las siguientes funciones:

- **command_parse**: Se encarga de parsear una cadena de caracteres recibida.
- **command_get_args**: Retorna el valor de DATA del comando.
- **command_operation**: Retorna qué operación se quiere realizar.
- **command_calc_checksum**: Esta función de uso interno calcula el checksum del comando recibido.
- **command_response**: Retorna el comando de respuesta a un comando recibido.

En la figura 4 podemos ver el diagrama de flujo con su funcionamiento.

2.4. Sound

Este módulo se encargará de apagar, prender y generar el sonido a una frecuencia fija que esté dentro del rango permitido que es de 10 a 1000Hz. Para esto este módulo se apoya en el módulo TPM (que se describira en la siguiente sección). Las funciones de este módulo son las siguientes:

- **sound_init**: Inicializa el modulo.
- **sound_on**: Enciende el sonido.
- **sound_off**: Apaga el sonido.
- **sound_set_fre**: Permite establecer la frecuencia.
- **sound_int_handler**: Manejador de interrupción del TPM.

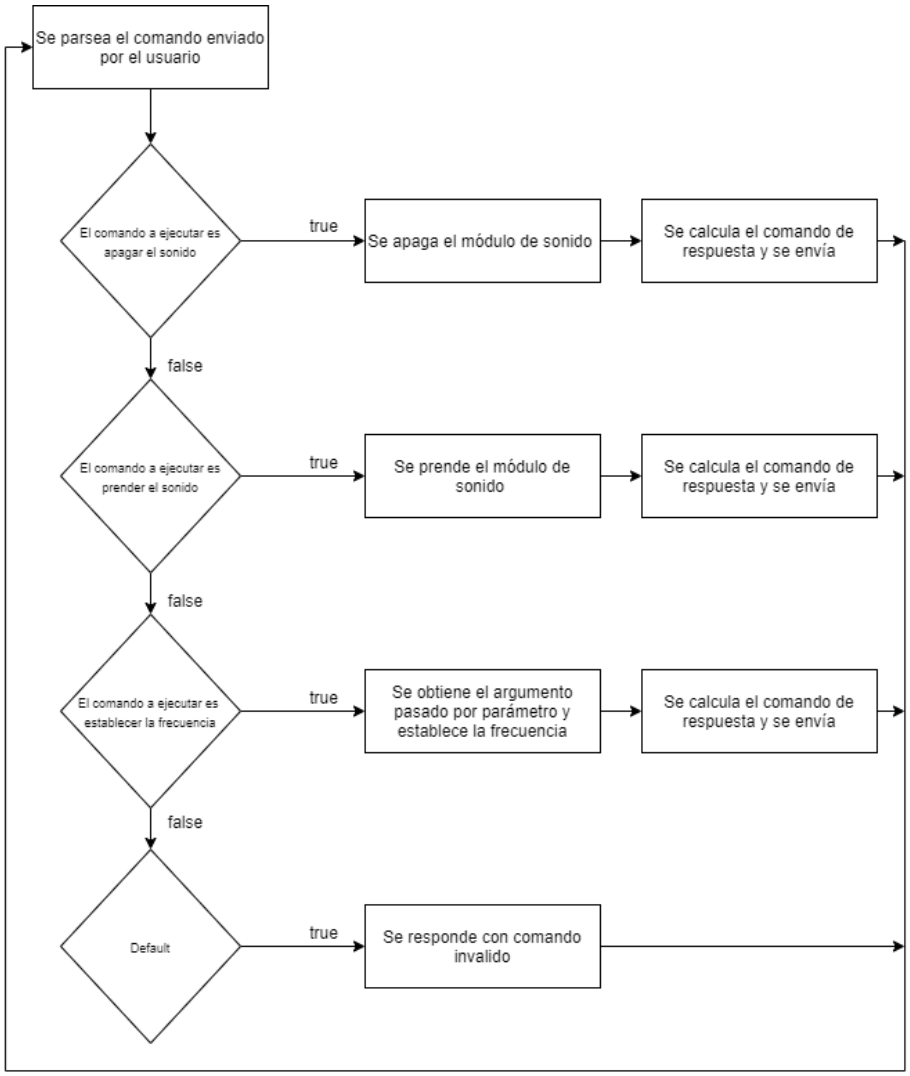

Figura 4: Diagrama de flujo correspondiente a la utilización del modulo Command.

2.4.1. TPM (Timer Pulse-Width Modulation)

Para la generación de sonidos, se utiliza el módulo TPM (Timer Pulse-Width Modulation). El microcontrolador posee dos de estos módulos exactamente iguales. El TPM sirve para temporizar interrupciones, generar ondas PWM o hacer mediciones de tiempo entre eventos proveniente de periféricos.

Cada TPM está formado por un contador de 16 bits (TPMxCNT) que se incrementa continuamente a una frecuencia que puede ser: la frecuencia del clock de bus predividida, clock fijo de sistema o un clock externo. En este caso, se utiliza el clock de bus con un valor establecido en el predivisor del TPM.

Ademas el TPM tiene dos canales, cada uno asociado a un pin, que permiten, entre otras cosas, hacer un cambio en el nivel lógico de su pin cuando el contador del TPM (TPMxCNT) coincide con un valor propio del canal (TPMxCnV), disparando una interrupción manejable por el software.

2.4.2. Onda de frecuencia fija

Para generar la onda cuadrada de frecuencia fija, se utiliza el TPM1, con su canal 0 asociado al pin PTC0. El canal 0 está configurado en modo Output Compare, de manera que cuando el contador del TPM1 coincida con el valor del canal, se hace un toggle de nivel lógico en el pin PTC0.

Para controlar con precisión la frecuencia a la que se cambia de nivel lógico la señal, se emplea una técnica en la que cada vez que ocurre una interrupción de canal, es decir, TPM1CNT es igual a TPM1C0V, se le suma a TPM1C0V un número entero positivo N_c. Este número es el que va a controlar la frecuencia con la que la señal cambia de valor. El N_c para que el evento ocurra a una frecuencia dada f está dado por:

$$N_c = \frac{f_c}{f \cdot p}$$

donde f_c es la frecuencia del bus, p es el predivisor elegido, que puede ser 1, 2, 4, 8, 16, 32, 64 o 128.

Como un ciclo de la onda cuadrada tiene en realidad dos cambios de nivel, se debe doblar la frecuencia a la que se hace el cambio, con lo que la fórmula del N_c que se utiliza es:

$$N_c = \frac{f_c}{2 f_{oc} p} \tag{1}$$

donde f_{oc} es la frecuencia deseada de la onda cuadrada. Se puede visualizar esto en la figura 5.

```
1  Funcion sound_int_handler()
2          Incrementar TPM1C0V en Nc
3          Reconocer interrupción
4  Fin sound_int_handler()
```

Pseudocódigo del manejador para generar la onda cuadrada.

2.4.3. Configuración TPM

Para configurar el módulo TPM primero elegimos la fuente de reloj, utilizaremos bus clock para nuestro caso con un predivisor en 4 con un módulo en cero para que cuente hasta el máximo. Habilitamos un canal 0 del TPM1 y elegimos el modo Output Compare y la acción en la salida elegimos que haga toggle, en la configuración del canal seleccionar el pin de salida y habilitamos por último la interrupción del canal 0. La descripción anterior se puede observar en la figura 6.

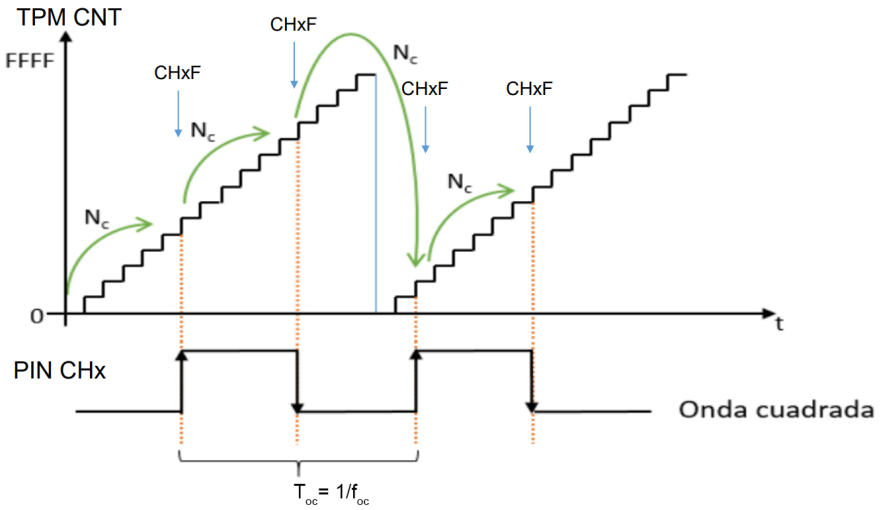

Figura 5: Generación de la onda cuadrada a una frecuencia fija utilizando un valor de N_c.

Para analizar el predivisor a utilizar para el rango de frecuencia de 10hz a 1000 Hz generamos una tabla utilizando la fórmula 1 para los siguientes valores.

Primero analizamos para 10 Hz, se puede ver en la figura 7 que el predivisor debe ser como mínimo de 8 para delante ya que menor a ese valor no podemos generar una frecuencia de 10Hz.

Segundo analizamos para 100 Hz, se puede ver en la figura 8 que cualquier predivisor genera una resolución relativa menor al 1 % por lo tanto se puede utilizar cualquier predivisor.

Por último analizamos para 1000 Hz, se puede ver en la figura 9 que con un predivisor menor de 8 podemos generar 1000 Hz con resolución relativa menor al 0.01 %.

Como conclusión podemos decir que para el rango de 10 a 15 Hz podemos utilizar un predivisor de 8 y para el rango de 16 a 1000 Hz un predivisor de 4. El registro que permite cambiar el predivisor en el microcontrolador posee 3 bits a establecer para definir el predivisor, se puede ver en la figura 10. Por lo tanto al estar trabajado con una frecuencia menor a 16 Hz establecemos el predivisor de 8 mientras que si trabajamos con frecuencias igual a 16 Hz hasta 1000 Hz establecemos el predivisor en 8. Como por default el módulo de sonido tiene una frecuencia de 400 Hz (por si el usuario prende el módulo de sonido ya tenga una frecuencia a reproducir por default) entonces se configuró el predivisor por defaul en 4.

3. Validación

Para verificar el funcionamiento mostraremos 3 situaciones que son las siguientes: el usuario ingresando un comando invalido, ingresando un comando válido que va ser de establecimiento de la frecuencia a setear e ingresando un comando fuera de rango de la frecuencia a establecer.

En este primer caso el usuario ingresa un comando no válido como el de la figura 16.

En donde el sistema responde “Comando invalido.” en formato hexadecimal, se puede ver en la figura 17.

Luego pasamos a establecer la frecuencia en la que queremos que se reproduzca el sonido una vez que se prenda, para eso elegí una frecuencia de 447 Hz que en formato hexadecimal son dos bytes:

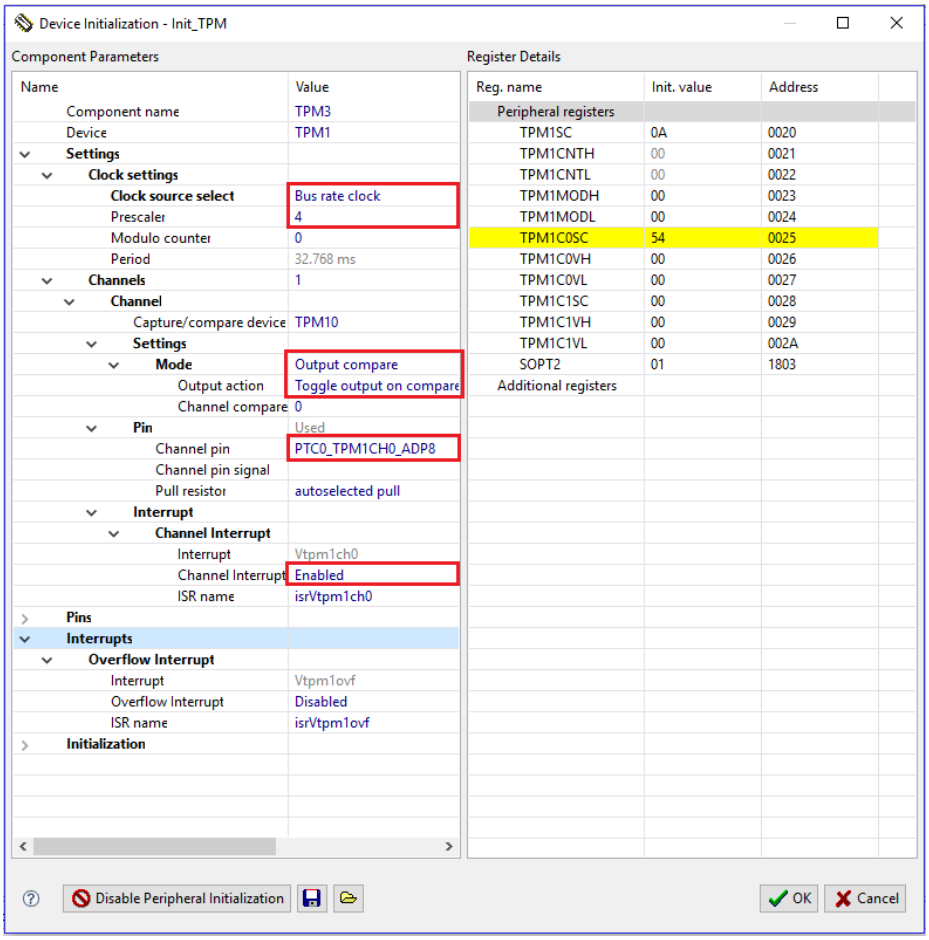

Figura 6: Configuración del TPM.

Predivisor	fmin [Hz]	fmax [Hz]	Nc@10Hz	delta(f)/f [%] @10 Hz
1	61	97861*	4M	-
2	30.5	97861*	2M	-
4	15.26	97861*	1M	-
8	7.63	97861*	50k	0.002
16	3.81	97861*	25k	0.004
32	1.9	97861*	12.5k	0.008
64	0.953	62500**	6250	0.016
128	0.477	31250**	3125	0.032

* limitada por la latencia de la interrupción

** limitada por la resolución de 1 TPMCLK

Figura 7: Tabla para un frecuencia de 10 Hz.

Predivisor	fmin [Hz]	fmax [Hz]	Nc@100Hz	delta(f)/f [%] @100 Hz
1	61	97861*	40k	0.0025
2	30.5	97861*	20k	0.005
4	15.26	97861*	10k	0.01
8	7.63	97861*	5k	0.02
16	3.81	97861*	2.5k	0.04
32	1.9	97861*	1250	0.08
64	0.953	62500**	625	0.16
128	0.477	31250**	312	0.32

* limitada por la latencia de la interrupción

** limitada por la resolución de 1 TPMCLK

Figura 8: Tabla para un frecuencia de 100 Hz.

Predivisor	fmin [Hz]	fmax [Hz]	Nc@1000Hz	delta(f)/f [%] @1000 Hz
1	61	97861*	4k	0.025
2	30.5	97861*	2k	0.05
4	15.26	97861*	1k	0.1
8	**7.63**	**97861***	**500**	**0.2**
16	**3.81**	**97861***	**250**	**0.4**
32	**1.9**	**97861***	**125**	**0.8**
64	**0.953**	**62500****	**62**	**1.6**
128	**0.477**	**31250****	**31**	**3.2**

* limitada por la latencia de la interrupción

** limitada por la resolución de 1 TPMCLK

Figura 9: Tabla para un frecuencia de 1000 Hz.

Table 16-5. Prescale Factor Selection

PS2:PS1:PS0	TPM Clock Source Divided-by
000	1
001	2
010	4
011	8
100	16
101	32
110	64
111	128

Figura 10: Registro a configurar para cambiar el predivisor a utilizar.

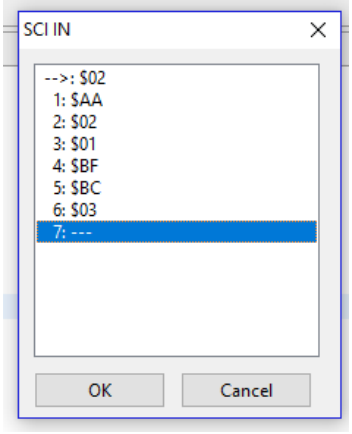

Figura 11: Ingreso de comando no valido.

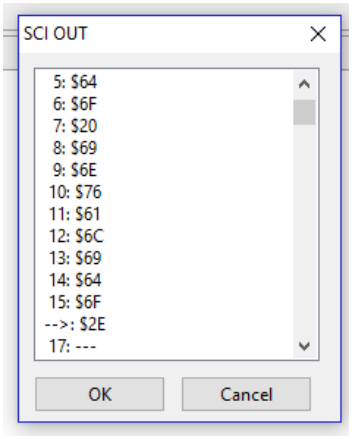

Figura 12: Respuesta de comando no valido.

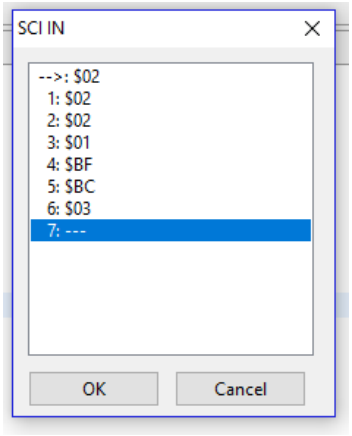

Figura 13: Ingreso de comando para establecer la frecuencia en 447Hz.

0x01 0xBF. Podemos ver el formato del comando en la figura 13.

Luego utilizamos el comando de encender el módulo de sonido, para que empiece a funcionar el módulo TPM. Podemos notar que al utilizar una frecuencia de 447 Hz utilizando las fórmulas vistas en la sección anterior obtenemos un N_c=2237 para ver si logramos la frecuencia a la cual queremos establecer, utilizamos el Cycle Counter que nos brinda el simulador del microcontrolador, podemos ver en la figura 14 que obtenemos un valor muy cercano el cual es un N_c=2235 que si aplicamos la fórmula obtenemos una frecuencia de 447,32 Hz. Por lo tanto logramos establecer la frecuencia en el valor definido en el comando con un pequeño margen de error.

La respuesta del sistema al comando anterior se puede ver en la figura 15.

Por último estableceremos una frecuencia fuera de rango que sera de 1520 Hz que en hexadecimal se expresaría como 0x05 0xF0 podemos ver en la figura 16 el comando enviado por el usuario para establecer esa frecuencia y en la figura 17 la respuesta dada por el sistema indicando que está fuera de rango la frecuencia.

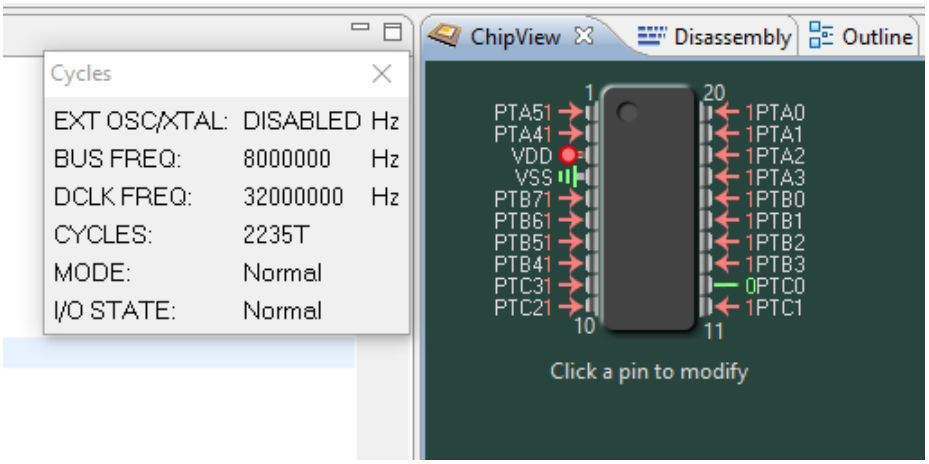

Figura 14: Verificación de que el TPM este funcionando a la frecuencia establecida.

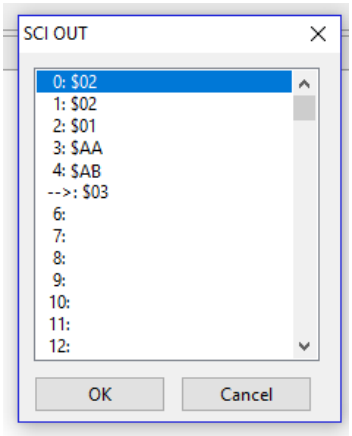

Figura 15: Respuesta al comando de establecimiento de frecuencia en 447 Hz.

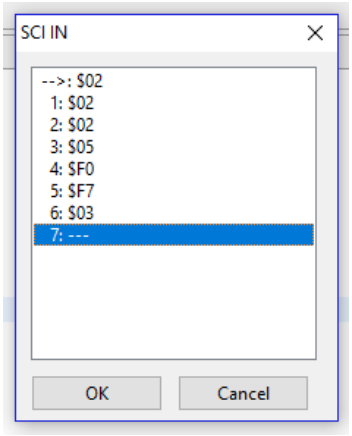

Figura 16: Ingreso de comando con frecuencia fuera de rango.

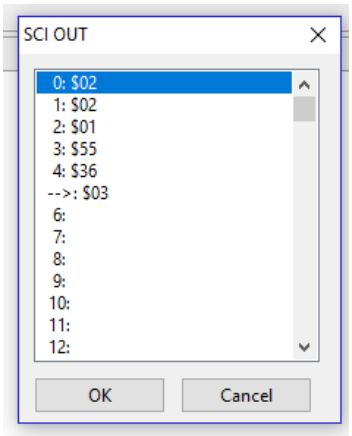

Figura 17: Respuesta de comando con frecuencia fuera de rango.

4. Apéndice: Código fuente

Módulos de la sección entrada salida

4.1. buffer.h

```
1  #ifndef BUFFER_H_
2  #define BUFFER_H_
3  #include "types.h"
4
5  #define BUFFER_CAPACIDAD 32
6  #define BUFFER_MOD_MASK 0x1F
7
8  struct buffer {
9          uint8_t size;
10         uint8_t head;
11         uint8_t tail;
12         uint8_t array[BUFFER_CAPACIDAD];
13 };
14
15 void buffer_init(volatile struct buffer *);
16
17 void buffer_push(volatile struct buffer *, uint8_t val);
18
19 uint8_t buffer_pop(volatile struct buffer *);
20
21 uint8_t buffer_empty(volatile struct buffer *);
22
23 uint8_t buffer_full (volatile struct buffer *);
24
25 #endif /* BUFFER_H_ */
```

4.2. buffer.c

```
1  #include "io/buffer.h"
2  #include "types.h"
3
4  /*
5   * Inicializa un buffer.
6   * Parametro:
7   * b: buffer a inicializar.
8   */
9  void buffer_init(volatile struct buffer *b){
10         b->size = 0;
11         b->head = 0;
12         b->tail = 0;
13 }
14
```

```
15  /*
16   * Agrega un elemento a la cola.
17   * Parametro:
18   * caracter: Caracter a agregar.
19   */
20  void buffer_push(volatile struct buffer *b, uint8_t val){
21          b->array[b->head] = val;
22          if (++(b->size) > BUFFER_CAPACIDAD) {
23                  b->size = BUFFER_CAPACIDAD;
24                  b->tail = (b->tail + 1) & BUFFER_MOD_MASK;
25          }
26          b->head = (b->head + 1) & BUFFER_MOD_MASK;
27  }
28
29  /*
30   * Saca el primer elemento agregado a la cola.
31   * Devuelve: El primer caracter agregado a la cola.
32   */
33  uint8_t buffer_pop(volatile struct buffer *b){
34          uint8_t val;
35          b->size--;
36          val = b->array[b->tail];
37          b->tail = (b->tail + 1) & BUFFER_MOD_MASK;
38          return val;
39  }
40
41  /*
42   * Comprueba si el buffer esta vacio.
43   * Devuelve: Valor uno si esta vacia, cero caso contrario.
44   */
45  uint8_t buffer_empty(volatile struct buffer *b){
46          return !b->size;
47  }
48
49  /*
50   * Comprueba si el buffer esta lleno.
51   * Devuelve: Valor uno si esta compelto el buffer, cero caso contrario.
52   */
53  uint8_t buffer_full(volatile struct buffer *b){
54          return b->size == BUFFER_CAPACIDAD;
55  }
```

Módulos de la sección Lib

4.3. serial.h

```
1  #ifndef SERIAL_H_
2  #define SERIAL_H_
3  #include "types.h"
4
5  void serial_init(void);
```

```
6
7  void serial_send(const char * str);
8
9  void serial_recv(char *str, const uint8_t n);
10
11 void serial_tx_handler(void);
12
13 void serial_rx_handler(void);
14
15 #endif /* SERIAL_H_ */
```

4.4. serial.c

```
1  #include "lib/serial.h"
2  #include "io/buffer.h"
3  #include "types.h"
4  #include <mc9s08sh8.h>
5
6  /* Flags de recepcion de datos */
7  static volatile char rx = 0;
8
9  /* Buffers utilizados para recepcion y transmision de datos */
10 static volatile struct buffer buffer_rx;
11 static volatile struct buffer buffer_tx;
12
13 /* Inicializacion de los buffers de transmision y recepcion */
14 void serial_init(){
15         buffer_init(&buffer_rx);
16         buffer_init(&buffer_tx);
17 }
18
19 /*
20  * Permite enviar una cadena de caracteres de forma serie.
21  * Parametro:
22  * buff: cadena de caracteres a enviar.
23  */
24 void serial_send(const char * buff){
25         SCIC2_TIE = 0;
26         while (*buff) {
27                 if (buffer_full(&buffer_tx)) {
28                         SCIC2_TIE = 1;
29                         while (SCIC2_TIE);
30                 }
31                 buffer_push(&buffer_tx, *(buff++));
32         }
33         SCIC2_TIE = 1;
34 }
35
36 /*
37  * Permite recibir una cadena de caracteres de forma serie e ir
```

```
38   * almacenadola en un buffer.
39   * Parametro:
40   * buff: buffer donde se almacenara la cadena recibida.
41   * n: tamanio de la cadena a recibir.
42   */
43  void serial_recv(char * buff, const uint8_t n){
44          uint8_t count = 0;
45          uint8_t car;
46
47          /* Se toma el control del buffer */
48          SCIC2_RIE = 0;
49
50          while (count < n ) {
51                  if (buffer_empty(&buffer_rx)) {
52                          rx = 0;
53                          SCIC2_RIE = 1;
54                          while(!rx);
55                          SCIC2_RIE = 0;
56                  }
57
58                  car = buffer_pop(&buffer_rx);
59                  *(buff++) = car;
60                  count++;
61
62                  /* Identifica si es una instruccion de 6 caracteres */
63                  if(car == 0x01 && count == 3)
64                          count++;
65  }
66
67          /* Se deja de tomar el control sobre el buffer */
68          SCIC2_RIE = 1;
69  }
70
71  /* Manejador de interrupcion a ejecutarse cada vez que el hardware dispone un
        caracter por puerto serial */
72  void serial_rx_handler(){
73          if (SCIS1_RDRF) {
74                  buffer_push(&buffer_rx, SCID);
75                  rx = 1;
76          }
77
78          if (buffer_full(&buffer_rx))
79                  SCIC2_RIE = 0;
80  }
81
82  /* Manejador de interrupcion a ejecutarse cada vez que el hardware puede mandar
        un caracter por puerto serie */
83  void serial_tx_handler(){
84          if (buffer_empty(&buffer_tx)) {
85                  SCIC2_TIE = 0;
86                  return;
87          }
```

```
88
89          if (SCIS1_TDRE)
90                  SCID = buffer_pop(&buffer_tx);
91  }
```

4.5. command.h

```
1   #ifndef COMMAND_H_
2   #define COMMAND_H_
3   #include "types.h"
4
5   #define COMMAND_OFF 0
6   #define COMMAND_ON 1
7   #define COMMAND_FREQ_SET 2
8   #define COMMAND_INVALID 0xFF
9
10  void command_parse(const char * line);
11
12  void command_get_args(uint16_t *);
13
14  int8_t command_operation(void);
15
16  int8_t command_calc_checksum(void);
17
18  int8_t command_response(uint8_t * line);
19
20  #endif
```

4.6. command.c

```
1   #include "lib/command.h"
2
3   /* Defino la estructura de un comando */
4   struct command
5   {
6           uint8_t stx;
7           uint8_t op;
8           uint8_t len;
9           uint8_t data[2];
10          uint8_t chks;
11          uint8_t etx;
12
13  };
14
15  /* Defino la variable en donde se almacenara el comando recibido */
16  volatile static struct command receive_command;
17
```

```
18 /*
19  * Parsea el arreglo recibido por parametro, si no cumple con la
20  * estructura se establece en el comando recibido la operacion como invalida.
21  * Parametro:
22  * line: una arreglo que representa un comando recibido.
23  */
24 void command_parse(const char * line){
25
26         receive_command.op = COMMAND_INVALID;
27
28         if(line[0]==0x02){
29                 if(line[2]==0x01 && line[5]==0x03){
30                         receive_command.stx = line[0];
31                         receive_command.op = line[1];
32                         receive_command.len = line[2];
33                         receive_command.data[0] = line[3];
34                         receive_command.chks = line[4];
35                         receive_command.etx = line[5];
36                 }
37                 if(line[2]==0x02 && line[6]==0x03){
38                         receive_command.stx = line[0];
39                         receive_command.op = line[1];
40                         receive_command.len = line[2];
41                         receive_command.data[0] = line[3];
42                         receive_command.data[1] = line[4];
43                         receive_command.chks = line[5];
44                         receive_command.etx = line[6];
45                 }
46         }
47 }
48
49
50 /*
51  * Evalua el comando recibido y retorna a que operacion corresponde.
52  * Retorna:
53  * El valor del comando a ejecutar.
54  */
55 int8_t command_operation(){
56         switch (receive_command.op) {
57                 case 0x01:
58                         return COMMAND_ON;
59                 case 0x02:
60                         return COMMAND_FREQ_SET;
61                 case 0x03:
62                         return COMMAND_OFF;
63                 default:
64                         return -1;
65         }
66 }
67
68 /*
69  * Convierte el dato del comando a un valor int.
```

```
70   * Retorna:
71   * El valor del dato en el comando como un valor tipo int.
72   */
73  int16_t data_to_int(){
74          if(receive_command.len == 0x01){
75                  return receive_command.data[0];
76          }
77          return receive_command.data[0]*256+receive_command.data[1];
78  }
79
80
81  /* Retorna el valor del paramtros que es enviado en el comando */
82  void command_get_args(uint16_t * args){
83          *args = data_to_int();
84  }
85
86  /*
87   * Realiza el calculo del Checksum.
88   * Retorna:
89   * El valor del calculo del checksum.
90   */
91  int8_t command_calc_checksum(){
92          int8_t checksum = receive_command.stx^receive_command.op^
                receive_command.len^receive_command.data[0];
93          if(receive_command.len == 0x02)
94                  checksum ^= receive_command.data[1];
95          return checksum;
96  }
97
98  /*
99   * Establece el formato de la respuesta que se da al comando recibido.
100  * Parametro:
101  * line: arreglo que representa la instruccion de respuesta al comando
102  * ingresado por el usuario.
103  */
104 int8_t command_response(uint8_t * line){
105         line[0] = 0x02;
106         line[1] = receive_command.op;
107         line[2] = 0x01;
108         if(command_calc_checksum()== receive_command.chks){
109                 line[3] = 0xAA;
110                 if(data_to_int() < 10 || data_to_int() > 1000){
111                         line[3] = 0x55;
112                 }
113         }else{
114                 line[3] = 0xFF;
115         }
116         line[4] = line[0]^line[1]^line[2]^line[3];
117         line[5] = 0x03;
118 }
```

4.7. sound.h

```
1  #ifndef SOUND_H_
2  #define SOUND_H_
3  #include "types.h"
4
5  void sound_init(void);
6
7  void sound_on(void);
8
9  void sound_off(void);
10
11 char sound_set_freq(const uint16_t f);
12
13 void sound_int_handler(void);
14
15 #endif
```

4.8. sound.c

```
1  #include <mc9s08sh8.h>
2  #include "lib/sound.h"
3  #include "types.h"
4
5
6  #define PRE_TPM1_4 4
7  #define PRE_TPM1_8 8
8
9  #define MIN_FREQ 10
10 #define MAX_FREQ 1000
11 #define DEFAULT_FREQ 400
12
13 static uint16_t nc;
14
15 /* Inicializa el modulo TPM. Se establece la frecuencia por default a 400 Hz */
16 void sound_init(){
17         sound_set_freq(DEFAULT_FREQ);
18         sound_off();
19 }
20
21 /* Enciende el sonido */
22 void sound_on(void){
23         TPM1C0SC_CH0IE = 1;
24         TPM1C0SC_ELS0A = 1;
25         TPM1C0SC_ELS0B = 0;
26 }
27
28 /* Apaga el sonido */
29 void sound_off(void){
```

```
30          TPM1C0SC_CH0IE = 0;
31          TPM1C0SC_ELS0A = 0;
32          TPM1C0SC_ELS0B = 1;
33  }
34
35  /*
36   * Permite establecer la frecuencia.
37   * Parametro:
38   * f: valor de la frecuencia a realizar.
39   * Retorna:
40   * Si esta fuera de rango la recuencia 0, sino el valor 1.
41   */
42  char sound_set_freq(const uint16_t f){
43          if (f < MIN_FREQ || f > MAX_FREQ)
44                  return 0;
45          if(f<=15){
46                  TPM1SC_PS0 = 1;
47                  TPM1SC_PS1 = 1;
48                  TPM1SC_PS2 = 0;
49                  nc = 4000000UL/(PRE_TPM1_8*f);
50          }else{
51                  TPM1SC_PS0 = 0;
52                  TPM1SC_PS1 = 1;
53                  TPM1SC_PS2 = 0;
54                  nc = 4000000UL/(PRE_TPM1_4*f);
55          }
56          return 1;
57  }
58
59  /* Manejador de interrupcion del TPM */
60  void sound_int_handler(){
61          TPM1C0V += nc;
62          TPM1C0SC;
63          TPM1C0SC_CH0F=0;
64  }
```

Programa principal

4.9. types.h

```
1   #ifndef TYPES_H
2   #define TYPES_H
3
4   typedef unsigned char uint8_t;
5   typedef unsigned int uint16_t;
6   typedef unsigned long uint32_t;
7
8   typedef signed char int8_t;
9   typedef signed int int16_t;
10  typedef signed long int32_t;
11
```

```
12 #endif
```

4.10. main.c

```
1  #include <hidef.h>
2  #include "derivative.h"
3  #include "lib/serial.h"
4  #include "lib/command.h"
5  #include "lib/sound.h"
6
7
8  void MCU_init(void);
9
10
11 #define STR_MSIZE 7
12 char str[STR_MSIZE], resp[6];
13 static uint16_t arg;
14
15 void main(void) {
16         serial_init();
17         MCU_init();
18         sound_init();
19         for (;;) {
20                 serial_recv(str, STR_MSIZE);
21                 command_parse(str);
22                 switch(command_operation()) {
23                         case COMMAND_OFF:
24                         sound_off();
25                         command_response(resp);
26                         serial_send(resp);
27                         break;
28                         case COMMAND_ON:
29                         sound_on();
30                         command_response(resp);
31                         serial_send(resp);
32                         break;
33                         case COMMAND_FREQ_SET:
34                         command_get_args(&arg);
35                         sound_set_freq(arg);
36                         command_response(resp);
37                         serial_send(resp);
38                         break;
39                         default:
40                         serial_send("Comando_invalido.");
41                 }
42
43         }
44 }
```