Universitat d'Alacant
Universidad de Alicante

Escuela Politécnica Superior

Arduino como DSP para procesamiento de audio

Procesadores digitales de señal

Francisco Javier Bleda Molina
Pablo González Carrizo

Contenido

1. Introducción

El uso de procesadores digitales de señal (**DSP**) está a la orden del día en diversos campos. Muchos de los dispositivos que usamos habitualmente, como pueden ser nuestro teléfono móvil o el router que tenemos en casa, llevan dentro un chip DSP, que realiza operaciones numéricas a muy alta velocidad, para así ser capaz de procesar diversas señales a tiempo real.

En este trabajo intentaremos abordar y analizar las posibilidades de un microcontrolador de uso general con muy bajo coste como es **Arduino**, en el procesamiento digital de señales. Concretamente, centraremos nuestra investigación en señales de audio.

2. Procesamiento digital de la señal

El procesamiento digital de la señal se distingue de otras áreas de la ciencia computacional por el tipo de datos que utiliza: señales. Estas señales son originadas a partir de sensores que recogen datos del mundo real (vibraciones sísmicas, temperatura, sonido, imágenes…) y deben ser previamente digitalizadas.

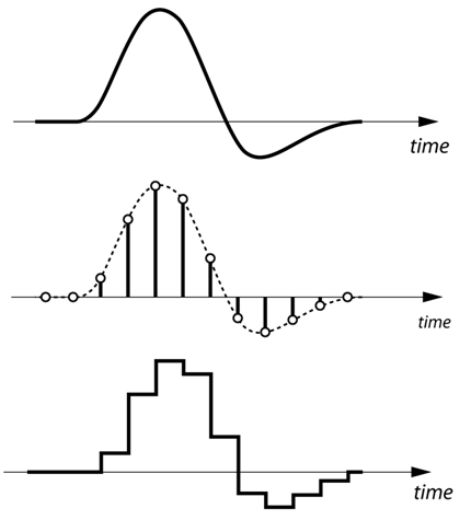

Figura 1: Señal analógica vs Señal digital

Este proceso de digitalización consta de tres fases: muestreo, cuantificación y codificación.

El **muestreo** o discretización temporal consiste en tomar muestras consecutivas de una señal analógica a una frecuencia constante. Para que el muestreo sea un proceso reversible y sin perdidas, es decir, poder recuperar la señal inicial a partir de la muestreada, la frecuencia utilizada debe ser como mínimo el doble de la frecuencia máxima de la señal, como así lo enuncia el **teorema de Nyquist**.

La **cuantificación** es la etapa donde se obtienen valores discretos de la amplitud de la señal previamente muestreada. Esta etapa es la que comprometerá la calidad de la señal digitalizada ya que necesitaríamos medir la amplitud del voltaje con un número de infinitos decimales para obtener el valor real de la amplitud de esta señal, esto implicaría almacenar un número infinito de bits.

Tras la cuantificación procedemos a la **codificación** donde a cada valor que previamente hemos obtenido le asignamos un número binario equivalente a los valores de tensión.

3. ¿Qué es un DSP?

Un procesador digital de señal (**DSP**) es un microprocesador con una arquitectura optimizada para cumplir las necesidades del tratamiento digital de la señal. Esta arquitectura nos permite realizar una gran cantidad de operaciones en muy poco tiempo, haciéndolo ideal para el tratamiento de señales en tiempo real, donde la latencia puede ser muy perjudicial.

Este tipo de dispositivos ha tenido un tremendo crecimiento en la última década, pudiéndolos encontrar en diversos dispositivos como teléfonos móviles, cámaras digitales, instrumentos médicos, módems inalámbricos y una larga lista de dispositivos que puedan tener relación con el procesamiento de las señales.

Aunque en un principio, se pueda pensar que su tarea podría ser realizada por otros microcontroladores de propósito general, esta es una idea totalmente equivocada. Steven W. Smith diferencia entre 2 tipos de operaciones principales realizadas por los ordenadores. [1]

Por un lado, encontramos las operaciones basadas en **tratamiento de información**, que son usadas comúnmente en procesadores de textos, administración de bases de datos, sistemas operativos u hojas de cálculo. Por otro lado, las otras operaciones realizadas por los computadores son las operaciones basadas en **cálculos matemáticos**. Las aplicaciones en las que se usan estas últimas son el tratamiento digital de la señal o las simulaciones de modelos de distintas ramas de la ciencia.

Más a bajo nivel, la diferencia entre estos dos tipos de operaciones radica en que, mientras que las operaciones de tratamiento de información se basan en movimiento de la información en memoria (***A*** --> ***B***) y comparación de valores (*if* ***A*** = ***B*** *then …*), en las operaciones de cálculos matemáticos destacan las sumas (***A*** + ***B*** = ***C***) y multiplicaciones (***A*** * ***B*** = ***C***). Aunque los computadores actuales sean capaces de realizar todas estas operaciones, el uso de sistemas operativos y la multitarea ha centrado su capacidad principalmente en la manipulación de datos, más que en los cálculos matemáticos. De ahí que existan los llamados **Procesadores Digital de Señal,** que se dedican exclusivamente a las operaciones de cálculos matemáticos. Una posible operación de este tipo, propia de un DSP puede ser aplicar un **filtro** **FIR**, cuya ecuación es del tipo:

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$

Como se puede ver, no es más que una combinación de retardos, sumas y multiplicaciones, lo que lo convierte en una operación idónea para ser realizada por un **DSP**.

3.1 Arquitectura

La arquitectura en la que están basados los ordenadores actuales es la arquitectura de **Von Neumann**. Esta arquitectura consiste en conectar permanentemente las distintas unidades que forman un computador, de forma que todo el ordenador está coordinado por un control central. Las instrucciones se almacenarán en una memoria. Esta arquitectura contiene una única memoria y un único bus que transfiere datos tanto de entrada como de salida de la unidad de procesamiento central. El funcionamiento de un ordenador se basa en la continua extracción de instrucciones de la memoria, su interpretación, la extracción de la memoria de los operandos implicados, el envío a la unidad encargada de realizar las operaciones y finalmente el cálculo del resultado.

Sin embargo, la arquitectura usada típicamente por los DSP es la **arquitectura Harvard**. Esta arquitectura utiliza memorias separadas para los datos y para las instrucciones de los programas, cada una con buses separados. Puesto que estos buses operan de forma independiente, las instrucciones de los programas y los datos pueden ser obtenidos a la vez. Esta arquitectura conlleva un aumento de la complejidad, pero permite obtener velocidades mayores que en las basadas en Von Neumann.

Además, también es común una arquitectura más sofisticada llamada **Súper Harvard Architecture** (SHARC). [2] Esta arquitectura es una modificacion sobre la arquitectura Harvard que introduce, entre otros muchos cambios, **un caché de instrucciones** en la unidad de procesamiento.

Un problema de la arquitectura Harvard es que el bus de la memoria de datos está mucho más

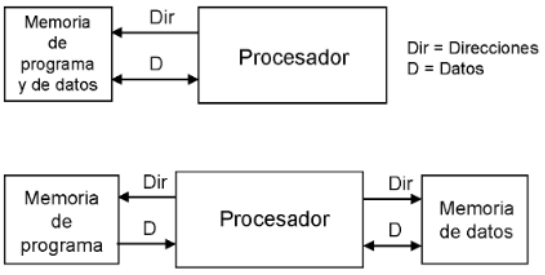

Figura 2: Arquitectura Von Neumann (arriba) y arquitectura Harvard (abajo)

ocupado que el de la memoria de programa. La mayoría de operaciones que realizamos requieren dos datos y una sola instrucción. Por

eso, SHARC propone utilizar parte de la memoria de programa para guardar datos.

Los DSP están continuamente realizando operaciones en bucles. Cada vez que se ejecuta un bucle tendremos que obtener las mismas instrucciones desde la memoria. Este comportamiento se puede optimizar, y por ello en la arquitectura SHARC se incluye un caché de instrucciones en la CPU. Este caché contendrá alrededor de 32 de las instrucciones utilizadas recientemente. Esto da lugar a que todo el traspaso de información necesario para una operación con una señal dentro de un bucle se pueda hacer en un único ciclo de reloj. Por ejemplo, en un filtro, la muestra de la señal de entrada vendrá del bus de la memoria de datos, el coeficiente vendrá del bus de la memoria de programa, y la instrucción vendrá de la memoria caché.

3.2 Formato aritmético y ancho de palabra

Una de las características más importantes que definen a un DSP es el tipo de formato y numero de bits de los datos que utiliza. Los DSP pueden operar con **números de coma fija** o **con números de coma flotante**.

Los números de coma fija destinan una cantidad fija de dígitos para la parte entera y otra para la parte fraccionaria. La cantidad de dígitos destinados a la parte fraccionaria indica la posición de la coma dentro del número.

En la representación en coma flotante se dividen los **n bits**, disponibles para representar un dato, en dos partes: una para la **mantisa M** (o fracción) y otra para el **exponente E**. Considerando que la mantisa tiene una longitud de p bits y que el exponente está formado por q bits. Se cumple que

$$n = p + q.$$

La mantisa contiene los dígitos significativos del dato, mientras que el exponente indica el factor de escala, en forma de una potencia de base 2.

Los DSP de coma flotante utilizan habitualmente un bus de datos de 32 bits. En los DSP de coma fija, el tamaño más común es de 16 bits. El tamaño del bus de datos tiene un gran impacto en el coste, ya que influye notablemente en el tamaño del chip y el número de patillas del encapsulado, así como en el tamaño de la memoria externa conectada al DSP.

Que un DSP tenga un tamaño de palabra de 16 bits no quiere decir que no pueda procesar una señal cuyas muestras ocupen, por ejemplo, 24 bits. Pero esto si tendrá un coste computacional y las operaciones necesitaran más ciclos que si usáramos un DSP de 24 o 32 bits.

De igual forma, que usemos un procesador de coma fija no quiere decir que no podamos realizar operaciones de aritmética de coma flotante. Pero estas deberán realizarse mediante software, resultando en un coste computacional mucho mayor.

3.3 Velocidad

Cuando queremos medir la velocidad de procesamiento de un DSP, normalmente se usa el tiempo de ciclo de instrucción, es decir, el tiempo necesario para ejecutar la instrucción más rápida del procesador. Una operación como, por ejemplo, la MAC (multiplicación y acumulación), tarda en ejecutarse en la mayoría de los DSP comerciales una única instrucción de reloj, por lo que saber cuánto tarda en ejecutarse una instrucción de este tipo nos hará tener una idea de cuánto tardaría en ejecutarse un algoritmo concreto. Sin embargo, esta medida no es más que una aproximación, puesto que existen muchas otras características que definirán la velocidad de procesamiento de un DSP. Instrucciones que en un DSP se hacen en un solo ciclo de reloj en otros pueden tardar varios ciclos.

3.4 Segmentación ("pipelining")

Pipelining es una técnica usada en la mayoría de DSP del mercado que consiste en dividir una secuencia de operaciones en otras más sencillas y ejecutar en la medida de lo posible cada una de ellas en paralelo. Se consigue por lo tanto reducir

tiempo total requerido para completar un conjunto de operaciones. Esta técnica es usada en muchos DSP.

4. El uso de DSP en audio

Tradicionalmente, los DSP han sido ampliamente utilizados en el mundo del audio para conseguir distintos efectos en la señal. Al comienzo del tratamiento digital de audio, los computadores requerían de dispositivos externos con el que poder procesar el sonido, ya que el propio procesador del ordenador no era capaz de realizar estas tareas, o al menos, no con la velocidad necesaria. Esto se debe a que estaban diseñados para un propósito general. La arquitectura de los DSP nos ha permitido obtener procesamientos avanzados con una menor latencia.

Las empresas encargadas de crear efectos de audio, con el fin de mejorar las mezclas y conseguir diferentes tipos de sonido recurrieron a estos dispositivos de hardware externos a la hora de implementar sus efectos. Además, otro motivo que las llevó a ello, es que el hardware no se puede copiar o falsificar tan fácilmente como un plugin software.

Una característica muy importante de los **DSP** que los diferencia de un computador estándar es que tenemos la capacidad de predecir cuánto tardará en ejecutarse una determinada serie de operaciones. Cuando copiamos una serie de ficheros de una carpeta a otra o cargamos una imagen o un documento, no sabemos exactamente cuántos milisegundos tardará el ordenador en realizar esta acción. Pero no necesitamos saberlo puesto que el que tarde unos milisegundos más o menos no afectará a la tarea que estamos realizando. Sin embargo, en un procesamiento de audio a tiempo real donde éste está muestreado a 48000 muestras por segundo, disponemos de (1 / 48000) = 0.02 milisegundos para procesar cada muestra. Como tardemos más de ese tiempo tendremos diversos efectos indeseables que pueden arruinar por completo el audio. Un DSP está diseñado para soportar tareas de altas prestaciones, repetitivas y numéricamente intensas.

5. Características Arduino

Arduino es una plataforma de **código abierto** usada para el desarrollo de proyectos de electrónica [3]. Arduino consta, por una parte, de una placa física que puede ser programada (refiriéndonos al microcontrolador) y por otra parte un software o IDE, que podemos correr en nuestros ordenadores y nos permite escribir código que correrá en el microprocesador.

Arduino se ha hecho popular entre diversos

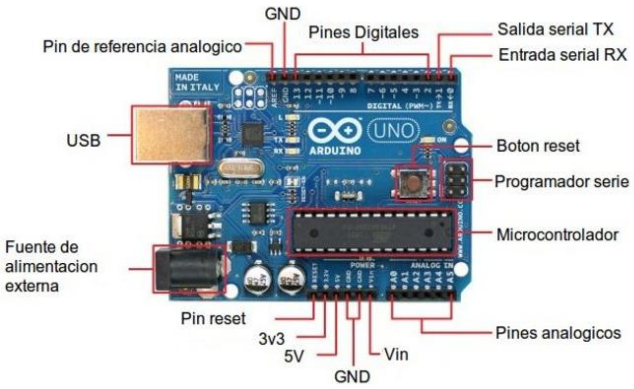

Figura 3: Placa de Arduino

perfiles de personas, que no tienen por qué tener altos conocimientos de electrónica o programación. Por otra parte, el IDE de Arduino utiliza una versión simple de **C++** siendo muy fácil el aprendizaje de su sintaxis.

Arduino Uno es el modelo más conocido, es el que trataremos en este documento. El corazón de Arduino Uno es el microprocesador **ATmega328** [4] de la marca Atmel. Este modelo consta de 14 pines digitales que pueden ser in/out y 6 de estos pines pueden ser usados como salidas PWM (que más adelante veremos para qué nos sirven). Cuenta también con 6 entradas analógicas, el conector USB, la entrada de corriente, un botón de reset y otros pines que veremos a continuación con más detalle.

Microcontrolador	Atmega328
Voltaje de operación	5V
Voltaje de entrada recomendado	7 – 12V
Voltaje de entrada límite	6 – 20V
Pines para entrada – salida digital	14 (6 pueden usarse como salida de PWM)
Pines de entrada analógica	6
Corriente continua por pin IO	40 mA
Corriente continua en el pin 3.3V	50mA
Memoria Flash	32kB (0.5 kB ocupados por el bootloader)
SRAM	2 kB
EPROM	1 kB
Frecuencia de reloj	16 MHZ

Figura 4: Principales características de Arduino UNO

5.1 Arquitectura de Arduino UNO

Este microcontrolador, al igual que los DSP, está basado en la **arquitectura Harvard**, donde el código del programa y los datos del programa se encuentran en memorias diferentes.

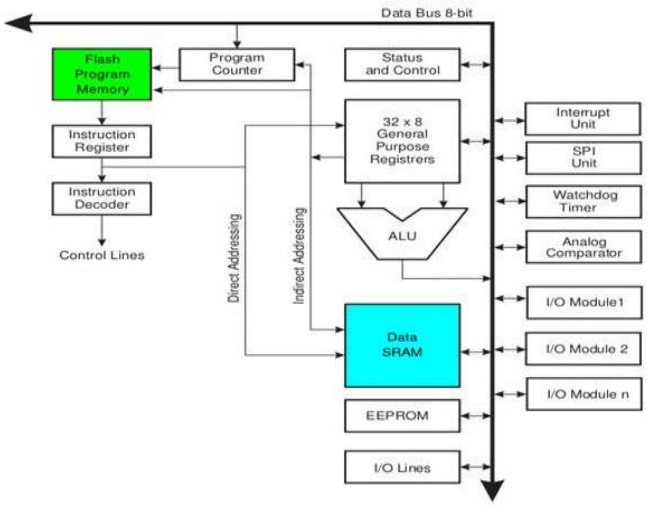

Figura 5: Arquitectura del Atmega328

El Atmega328 tiene 32kb de memoria flash, 2kb de SRAM 1kb, de EPROM y un reloj de 16MHz

5.2 Pines Digitales Inputs/Outputs

Los **14 pines digitales** pueden funcionar como entrada o salida. Cada uno de ellos puede dar o recibir hasta 40mA y tienen una resistencia de pull-up de 20-50KOhm. Por otra parte, también hay un par de pines dedicados a la transmisión de datos llamados Rx y Tx.

5.3 Entradas Analógicas

Arduino consta de **6 entradas analógicas**, A0-A5, con las cuales podemos obtener una resolución de 10bits $2^{10}\approx1024$ niveles. Por defecto tienen una tensión de 5V, pero esta puede ser cambiada utilizando el pin de AREF

5.4 Relojes

Varios son los relojes incluidos en la placa, que proporcionan diferentes señales a diferentes frecuencias destinadas a diferentes partes del controlador. Básicamente, estos relojes lo que hacen es enviar una señal cuadrada que proporciona la frecuencia a la que van a operar la CPU, ADC, el acceso a la memoria y otros componentes del microcontrolador.

Un concepto que debemos de conocer a la hora de utilizar los relojes, son los ***prescaler,*** cuyo fin es dividir las frecuencias de reloj, en fracciones de la frecuencia principal para obtener una frecuencia deseada.

El ***system clock*** o reloj del sistema nos va a proporcionar la frecuencia a la que va a funcionar el sistema. Otros relojes importantes son el *I/O* y el del ADC, usado para controlar la frecuencia de muestreo de la señal de entrada.

5.5 Registros e interrupciones

La CPU del Atmega328 consta de una unidad aritmeticológica que trabaja con 32 registros, porciones de memoria que proporcionan datos para el correcto flujo de funcionamiento de trabajo.

Por otra parte, las interrupciones en Arduino te permiten detectar eventos de forma asíncrona con independencia de las líneas de código que se estén ejecutando en ese momento en tu microcontrolador.

5.6 Timers / Contadores

Un timer es un registro en el cual tenemos un valor que se va incrementando con la señal de reloj. Cuando este alcanza su valor máximo, vuelve a 0 generando además una interrupción.

Arduino UNO tiene dos contadores de 8bit, y uno de 16 bits, cada uno puede ser configurado por separado.

5.7 Memoria

Arduino tiene 3 memorias las cuales pueden ser utilizadas para almacenar datos. Estas son las características de las memorias que utiliza

Nombre	Capacidad(KB)	Duración de datos	Ciclos de escritura
FLASH	32	SI	1
SRAM	2	NO	2
EEPROM	1	SI	30

Figura 6: Memorias de Arduino UNO

5.8 ADC

El conversor analógico-digital es el que permite realizar el proceso de digitalización de las señales analógicas que llegan a un determinado pin de Arduino. El ADC de Arduino utiliza un circuito de *Sample and Hold* que mantiene el voltaje de entrada constante hasta el final de la conversión. Esto fija un voltaje que es comparado con uno de referencia para obtener una aproximación de hasta 10 bits. Este tipo de conversión es conocido como **aproximaciones sucesivas.** Si requiriésemos de una conversión más rápida tendríamos que sacrificar la calidad y usar solo hasta 8bits. La conversión tarda entre 13-250 microsegundos, dependiendo de varios parámetros de configuración que influyen en la precisión del resultado.

El mecanismo del convertidor tiene un reloj dedicado para asegurar que la conversión se realiza independientemente de las otras partes del µC.

El ADC del Atmega328 tarda 13 ciclos de reloj en realizar la conversión. Sin embargo, el reloj usado en el ADC, viene de un **prescaler** que por defecto está configurado para dividir la frecuencia principal de 16MHz entre 128, siendo la velocidad del reloj a la entrada del ADC 16MHz/128 = 125KHz. Quedando, por lo tanto, una frecuencia de muestreo de 9.6kHz, lo que daría una frecuencia máxima para la señal de alrededor de 4KHz (muy similar a la de la línea telefónica). Mediante el registro ADCSRA podremos modificar el prescaler, en caso de querer tener una frecuencia de muestreo mayor, hasta un máximo teórico de 615KHz (Para una frecuencia de reloj de 8MHz). Sin embargo, esa frecuencia de muestreo es imposible de conseguir experimentalmente puesto que el comportamiento del ADC a esta frecuencia de muestreo es totalmente impredecible e inestable. Experimentalmente [5] se ha demostrado que usando una frecuencia de reloj de 1MHz, que da lugar a una frecuencia de muestreo de 77KHz, es el máximo valor que podemos conseguir para el ADC, teniendo una precisión de 8 bits.

5.9 PWM

Para volver a obtener una señal analógica, a partir de Arduino necesitamos un conversor digital-analógico (DAC), pero el microcontrolador no dispone de ninguno. Lo que haremos es simular una señal analógica utilizando señales digitales moduladas en ancho de pulso. Estas señales se conocen como señales PWM, *pulse width modulation*.

La amplitud de la señal que queremos generar modulará el ancho de los pulsos de una señal cuadrada digital. Si queremos obtener una amplitud de 5V, tendremos la señal PWM a 5V durante el 100% de la duración de su periodo. Si, por ejemplo, queremos obtener 2.5V, la señal PWM estará a 1 (5V) durante el 50% de la duración del periodo.

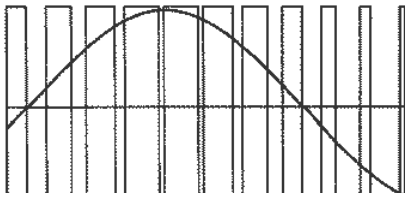

Figura 7: Señal analógica frente a señal PWM

Si esta señal PWM es filtrada adecuadamente con un filtro paso bajo, obtendremos la señal analógica deseada consiguiendo así quitar el ruido en alta frecuencia debido a la modulación y suavizando la señal.

6. Arduino como DSP: Síntesis de diferentes ondas

El primer ejemplo del que hablaremos es la síntesis de distintos tipos de ondas básicas mediante Arduino. Usaremos, igual que en el resto de ejemplos, una señal PWM para evitar tener que usar un DAC externo. En el pin por el que sacaremos la señal pondremos el jack de un cable cuyo otro extremo estará conectado directamente a una tarjeta de sonido externa conectada a nuestro PC. Así podremos ver la forma de onda que hemos generado, y escuchar posteriormente el resultado. Idealmente, deberíamos usar un filtro pasivo paso bajo formado por un condensador y una resistencia para filtrar la señal PWM y obtener los valores deseados. Sin embargo, lo que haremos nosotros es aplicar un filtro paso bajo a posteriori, mediante software, a la señal grabada. Además, como veremos, debido a la mala calidad de los componentes usados (tarjeta de sonido muy básica, y cables de mala calidad) la señal que obtiene el ordenador, se parece más a una señal típica analógica que a una señal PWM puesto que al pasar por estos dispositivos ha perdido las frecuencias altas. En el ATmega328 encontramos 3 timers: Timer0 y Timer2, que son timers de 8 bits y Timer1, que es de 16 bits (aunque puede ser configurado para trabajar a 8 bits). TCNT1, el registro usado en el Timer1 cuenta repetidamente desde 0 a 65535. Además, tiene el modo **fast PWM** que es el que usaremos. Cuando el contador se desborda, y vuelve a 0, la salida pasa a estado alto para indicar el principio de un nuevo ciclo. Cuando TCNT1 llegue al valor indicado por OCR1A su estado pasará a bajo.

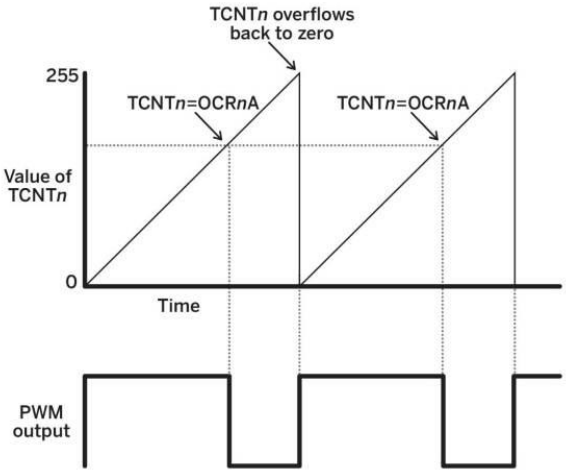

Figura 8: Generación de señal PWM mediante un timer

El código que usaremos es el siguiente:

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

/******** Parametros de la onda
********/
#define PI2 6.283185
#define AMP 127 // Amplitud de la
onda

#define OFFSET 128

/******** Lookup table ********/
#define LENGTH 256 //longitud del
array de muestras
byte wave[LENGTH]; // array de
muestras

void setup() {

/* Rellenamos el vector con las
muestras de una señal senoidal */
for (int i=0; i<LENGTH; i++) {
   float v =
(AMP*sin((PI2/LENGTH)*i)); //
Calculamos el valor
   wave[i] = int(v+OFFSET); //
Guardamos la muestra en el vector
 }

/****Configuramos el timer 1 para la
salida de la señal PWM ****/
 pinMode(9, OUTPUT); // Ponemos el
pin 9 en modo salida
 TCCR1B = (1 << CS10); //Configuramos
el prescaler para que no divida a la
señal de 16MHz
 TCCR1A |= (1 << COM1A1); // Nivel
bajo para el pin cuando TCNT1=OCR1A
```

```
 TCCR1A |= (1 << WGM10); // 8-bit
fast PWM mode
 TCCR1B |= (1 << WGM12);

/******** Configuramos timer 2
********/
 TCCR2A = 0; //
 TCCR2B = (1 << CS21); //Configuramos
el prescaler para que divida entre 8
 TIMSK2 = (1 << OCIE2A); // Creamos
interrupción cuando TCNT2 = OCRA2
 OCR2A = 32; // Configuramos la
frecuencia de la onda generada
 sei(); //Habilitamos las
interrupciones
}

void loop() {
}

/******** Se llama cada vez que
TCNT2 = OCR2A ********/
ISR(TIMER2_COMPA_vect) {
 static byte index=0;
 OCR1AL = wave[index++]; //
Actualizamos la salida PWM
 TCNT2 = 4; //Compensamos el tiempo
utilizado en atender a la
interrupción
}
```

Comenzamos calculando las muestras que equivaldrían a un ciclo completo de la forma de onda que queremos conseguir, y guardando éstas en un vector. A continuación, configuramos el **timer1** para que genere una señal pwm usando el modo **fast pwm**, y lo configuramos para que trabaje a 8 bits y no a 16.

Mediante el **timer2** vamos actualizando el valor del registro **OCR1A** con cada muestra de las calculadas anteriormente. Esto se realizará en el *interrupt service routine.* Mediante **OCR2A** controlaremos la velocidad a la que le pasamos una nueva muestra a la señal PWM, siendo capaces por lo tanto de configurar mediante este parámetro la frecuencia de la onda. Teniendo en cuenta que hemos configurado para este timer un prescaler de 8, **TCNT2** se actualizará a una frecuencia de 2MHz. Para calcular la frecuencia de la onda generada utilizaremos la siguiente ecuación:

$$f = \frac{Frec\ de\ actualización\ de\ TCNT}{Valor\ OCR2A\ * Tam\ \ vect\ muestras}$$

Tras esto, ponemos el valor de **TCNT2** a 4 para compensar el tiempo que ha tardado en ejecutarse la interrupción.

Mediante distintos algoritmos en la generación del vector de muestras podremos generar distintos tipos de ondas (cuadradas, triangulares, rampas…) manteniendo el resto del código igual. Incluso podremos realizar síntesis aditiva sumando, en la creación del vector, varias muestras de distintas ondas.

Además, podríamos prescindir de la generación del vector de muestras en tiempo de ejecución, y escribir en el propio código todos los valores necesarios. Para ello, habrá que usar la memoria de programa, puesto que con la memoria de datos no será necesario. Para ello crearemos el array de muestras con la siguiente instrucción:

```
const char wave[256] PROGMEM =
{85,75,25,20,45,…};
```

Para acceder a las muestras de este vector necesitaremos usar la función ***pgm_read_byte***.

7. Arduino como DSP: Librería Mozzi

En este segundo ejemplo mostraremos una librería donde se explota al máximo el tratamiento de audio en Arduino. La librería se llama **Mozzi** [6] y aunque está centrada en la síntesis de audio, varios de sus ejemplos pueden usarse perfectamente en el tratamiento de las muestras que reciba directamente del ADC. Entre las características de la librería se pueden destacar las siguientes:

- Frecuencia de muestreo de 16384Hz. Podemos obtener, por lo tanto, una frecuencia de señal máxima de 8KHz.
- Experimentalmente, es posible conseguir una frecuencia de muestreo de 32768Hz, aunque esta característica aún no es estable y puede dar problemas
- Posibilidad de muestras a 8 bits o a 14 bits (solo en muestras sintetizadas).

- Gran cantidad de osciladores, samples, envolventes y efectos
- Cantidad de ejemplos con diferentes parámetros modificables en tiempo real mediante sensores.
- Open source y extensible

Un sketch básico de un programa que haga uso de la librería Mozzi es el siguiente:

```
#include <MozziGuts.h>

void setup() {
  startMozzi();
}

void updateControl(){
  // your control code
}

int updateAudio(){
  // your audio code which returns
an int between -244 and 243
}

void loop() {
  audioHook();
}
```

En la función **setup()**debemos llamar a **startMozzi()** pasándole un valor que representará la frecuencia de actualización de las señales de control.

Dentro de la función **updateControl()** actualizaremos los parámetros necesarios de las señales que estemos usando en nuestro programa. Cualquier control de cualquier parámetro por medio de sensores o botones se realizará en esta función.

Dentro de **updateAudio()** es donde llevaremos a cabo la síntesis del audio. Puesto que esta función se ejecuta 16384 veces por segundo, disponemos de 1/16384 = 61 microsegundos para realizar todas las operaciones que necesitemos para calcular el valor de una muestra en concreto. Puesto que la frecuencia del procesador es de 16MHz, todas las operaciones que realicemos no pueden tardar más de 976 ciclos de reloj.

Mozzi usa 2 interrupciones distintas. Una para crear la señal de salida, a 16384Hz y otra a 64Hz (puede ser modificada) para actualizar los parámetros de control.

Simplemente añadiendo en la función **updateAudio**() cualquier código que genere valores de muestras entre -244 y 243 podremos sacarlas por el Pin9 de nuestro Arduino UNO, en forma de señal PWM.

Por ejemplo, mediante este código obtendremos a la salida, una señal de tipo seno a 440 Hz.

```
#include <MozziGuts.h>
#include <Oscil.h>
#include <tables/sin2048_int8.h>

#define CONTROL_RATE 128
Oscil <2048, AUDIO_RATE>
aSin(SIN2048_DATA);

void setup(){
  aSin.setFreq(440);
  startMozzi(CONTROL_RATE);
}

void updateControl(){
}

int updateAudio(){
  return aSin.next();
}

void loop(){
  audioHook();
}
```

Como vemos, Mozzi contiene una gran cantidad de funciones que nos permiten realizar estas tareas de forma más sencilla. Si por ejemplo queremos añadir una señal de control para producir un efecto tipo vibrato en el ejemplo anterior simplemente tendremos que añadir 3 líneas más a nuestro código.

Creamos el oscilador que controlará el vibrato y configuramos su frecuencia:

```
Oscil <2048, CONTROL_RATE> kVib(SIN2048_DATA);
kVib.setFreq(6.5f);
```

Solo nos quedará, en la función **updateControl()** donde modificamos los distintos parámetros de las señales usadas, añadir la frecuencia del vibrato a la frecuencia fija de 440Hz establecida para la señal.

```
void updateControl(){
  float vibrato = depth * kVib.next();
  aSin.setFreq(440.0+vibrato);
}
```

En la siguiente imagen se muestra el funcionamiento interno de esta librería.

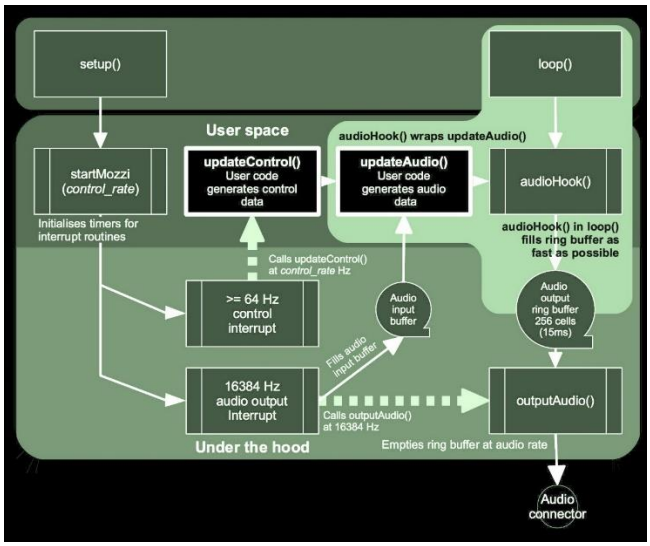

Figura 9: Funcionamiento de la librería Mozzi

Además, mediante la función **getAudioInput()** podremos obtener muestras directamente desde el ADC. Este sería una sketch vacío que serviría como plantilla para un procesamiento de las muestras de audio recibidas a través del Pin A0

```
/*  Test of audio input using Mozzi
sonification library.
   Tim Barrass 2013, CC by-nc-sa.
*/

//#include <ADC.h>  // Teensy 3.1
uncomment this line and install
http://github.com/pedvide/ADC
#include <MozziGuts.h>

void setup(){
  startMozzi();
}


void updateControl(){
}

int updateAudio(){
  int asig = getAudioInput(); //
range 0-1023
  asig = asig - 512; // now range
is -512 to 511
   // Apply here all the effects
you want.
  // output range in STANDARD mode
is -244 to 243,
  // so you might need to adjust
your signal to suit
  return asig;
}

void loop(){
  audioHook();
}
```

Usando el objeto **LowPassFilter** podríamos, por ejemplo, realizar un filtro paso bajo de la señal de entrada.

```
LowPassFilter lpf;
lpf.setCutoffFreq(200);


int updateAudio(){
  int asig = getAudioInput()-512;
  asig = lpf.next(asig>>1);
  return asig;
}
```

Entre la larga lista de los efectos implementados por la librería que podemos realizar mediante métodos casi iguales al nombrado anteriormente, están **delay, reverberación, flanger, chorus, tremolo y diversos filtros**.

Dentro de la síntesis de audio la librería proporciona también métodos para **la síntesis FM, AM**, por **tabla de ondas**, con la posibilidad de modificar las envolventes de forma muy sencilla.

Además, teniendo en cuenta que podemos acceder y modificar cada una de las muestras la lista de efectos es prácticamente infinita. Sin embargo, siempre tendremos que tener en cuenta la gran limitación que produce el tener una memoria para datos de únicamente 2KB, lo cual no nos permitirá tener buffer de muchas muestras. Esto hará que no podamos tener filtros de una gran cantidad de coeficientes.

8. Optimización del código

Uno de los motivos del éxito de Arduino con respecto a otros microcontroladores es su facilidad de uso. El uso de diversas librerías de alto nivel, facilita la tarea de desarrollar funcionalidades y nos abstrae y encapsula la dificultad de la tarea. Esto lo hace ideal para ser utilizado por personas que no tienen por qué conocer en profundidad la arquitectura del microprocesador, ni cientos de detalles de bajo nivel de los que ya se ocupan las librerías.

Sin embargo, este encapsulamiento de las distintas funcionalidades, en ocasiones da lugar a un código menos optimizado, y, por lo tanto, más lento. Por eso, en aplicaciones de alto rendimiento tendremos que intentar prescindir de estas librerías y conocer los distintos registros que encontramos en el procesador que lleva incorporado Arduino y su funcionamiento, para poder así escribir directamente sobre ellos, ahorrando varios ciclos de reloj que serán necesarios para poder conseguir el procesamiento en tiempo real.

Una lectura obligada es la guía de optimización de código para microcontroladores de 8 bits de Atmel [7].

Un claro ejemplo de lo explicado anteriormente es el uso de la función **digitalWrite** que permite poner a 1 o a 0 la salida de un pin concreto. Si miramos el código de esta función, esto es lo que realiza:

```
void digitalWrite(uint8_t pin,
uint8_t val)
{
    uint8_t timer =
digitalPinToTimer(pin);
    uint8_t bit =
digitalPinToBitMask(pin);
    uint8_t port =
digitalPinToPort(pin);
    volatile uint8_t *out;

    if (port == NOT_A_PIN) return;

    // If the pin that support PWM
output, we need to turn it off
    // before doing a digital write.
    if (timer != NOT_ON_TIMER)
turnOffPWM(timer);

    out = portOutputRegister(port);

    uint8_t oldSREG = SREG;
    cli();

    if (val == LOW) {
        *out &= ~bit;
    } else {
        *out |= bit;
    }

    SREG = oldSREG;
}
```

Todo este código, para asignar un valor a un pin tarda aproximadamente 50 ciclos en ejecutarse. Sin embargo, si nos fijamos en el datasheet del Atmega328P [8] vemos que contiene varios registros que nos permiten cambiar directamente el estado de los distintos pines. Debido al uso del compilador **GCC**, todas las variables que se especifican en el datasheet del Atmega328P para nombrar los diferentes registros se podrán usar directamente en el código en C++ de nuestro programa para Arduino Para poner a 1 un pin en concreto simplemente debemos ejecutar esta instrucción: *SET(PORTx, pin);* habiendo definido previamente el macro de la siguiente forma: *#define SET(x,y) (x|=(1<<y))*

De la misma forma, mediante *CLR(PORTB, pin) ;* con el marcro *#define CLR(x,y) (x&=(~(1<<y)))* podremos poner un pin a 0.

De esta forma el escribir sobre un pin concreto tardará aproximadamente 2 ciclos.

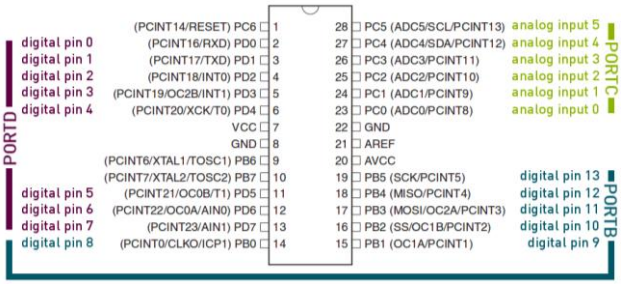

Figura 10: Pines del Atmega328P con los correspondientes registros que permiten modificarlos

Un factor muy importante a tener en cuenta es que el chip Atmega328p está diseñado para utilizar aritmética de punto fijo. Toda la aritmética de coma flotante se realizará por software, haciendo que esta sea mucho más lenta. Por lo tanto, siempre que sea posible intentaremos evitar el punto flotante en Arduino. Además, intentaremos elegir siempre los tipos de datos más pequeños, pero que nos aseguren que no se desbordarán cuando realicemos operaciones con ellos.

Además, las multiplicaciones y divisiones son un claro de cuello de botella en nuestro código. Utilizando operadores de aritmética de bits podremos realizar éstas en apenas 1 o 2 ciclos de reloj. Por ejemplo, para multiplicar o dividir entre 2 un número, únicamente tendremos que desplazar sus bits una posición hacia la izquierda o hacia la derecha respectivamente. Puesto que estas operaciones únicamente se pueden realizar de esta forma con múltiplos de 2, intentaremos usar potencias de 2 en todos los sitios posibles de nuestro código.

En ocasiones necesitaremos escribir en Arduino instrucciones directamente en ensamblador para conseguir una mayor velocidad en nuestro programa. Mediante la instrucción __asm__() podremos escribir instrucciones directamente en ensamblador, que Arduino será capaz de realizar. Un ejemplo es el uso de __asm__("nop\n\t"); cuando queremos conseguir delays muy cortos. La instrucción “nop” dura un único ciclo de reloj.

Como ya comentamos, Arduino usa, al igual que muchos DSP, una arquitectura Harvard. Esto implica que tiene distintas memorias para los programas que para los datos. Para ahorrar RAM, podemos utilizar la palabra PROGMEM para poder definir las variables dentro de la memoria de programa y así ahorrar memoria RAM. Al ser la memoria de los programas, una memoria Flash, podremos acceder a ella de forma

Además, como último recurso, Arduino nos permite añadir un nuevo oscilador de cristal que permitirá realizar operaciones

9. Conclusión

Mediante esta investigación hemos comprobado que la capacidad de Arduino queda muy lejos de la capacidad de procesamiento de los DSP comerciales, y es inviable sustituirlos por Arduinos. Sin embargo, puede ser una opción a tener en cuenta, con un coste muy reducido, en algunas aplicaciones que no requieran un rendimiento demasiado alto. Hemos comprobado que Arduino es más que un simple microcontrolador para principiantes en el mundo de la electrónica, sino que, con conocimientos sobre su arquitectura se pueden conseguir resultados tan sorprendentes como los de la librería Mozzi.

10. Referencias

[1] Steven W. Smith, “The Scientist an Engineer's Guide to Digital Signal Processing”, Capítulo 1, http://www.dspguide.com/ch28/1.htm [ONLINE]

[2] Analog Devices, SHARC Processors, http://www.analog.com/en/products/processors-dsp/sharc.html [ONLINE]

[3] Web de Arduino, https://www.arduino.cc [ONLINE]

[4] Atmel, “Atmega238p”, http://www.atmel.com/devices/atmega328p.aspx [ONLINE]

[5] apcmag, “Arduino’s analog-to-digital converter: how it works”, http://apcmag.com/arduino-analog-to-digital-converter-how-it-works.htm/ [ONLINE]

[6] Mr Sensorium, “Arduino Mozzi library”, http://sensorium.github.io/Mozzi/ [ONLINE]

[7] Atmel, “Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers”, http://www.atmel.com/images/doc8453. pdf [ONLINE]

[8] Atmel, “Atmega238P datasheet”, http://www.atmel.com/images/atmel-8271-8-bit-avr-microcontroller-atmega48a-48pa-88a-88pa-168a-168pa-328-328p_datasheet_complete.pdf [ONLINE]