Cika Electrónica

Nota de Aplicación: CAN-066
Título: **DSP con Ramtron VRS51L3074: efecto de sonido: flanger**
Autor: Sergio R. Caprile, Senior Engineer

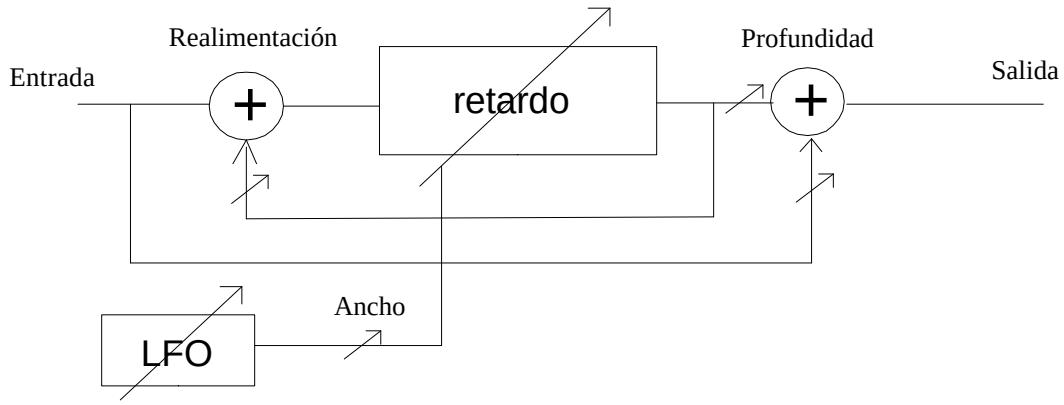
Revisiones	Fecha	Comentarios
0	12/02/07	

En esta nota de aplicación veremos un caso práctico de efectos de sonido con los Ramtron VRS51L3074. El marco de trabajo utilizado es el mismo de la CAN-065, y la aritmética es bastante compleja, por lo que se recomienda a los lectores no experimentados comenzar por dicha nota de aplicación.

Introducción

Gracias al esquema desarrollado en la CAN-063 podemos minimizar el tiempo ocioso y procesar en tiempo real una señal con un ancho de banda de poco más de 19KHz, quedándonos algo de tiempo para atender otras tareas.

Un efecto de sonido como el flanger responde a un diagrama en bloques como el siguiente:



Se trata de un retardo continuamente variable, controlado por un LFO que suele ser una señal triangular. Una parte de la señal así demorada se reinyecta a la entrada y otra se combina con la señal original para lograr un filtro peine variable, el cual va alterando el espectro del sonido original creando un efecto de "avión a chorro". La demora se obtiene ingresando muestras en un buffer circular y retirándolas unos intervalos después. Esto sólo provee determinados tiempos fijos de demora, por lo que para obtener tiempos intermedios se realiza una interpolación entre dos demoras sucesivas. La forma más simple y efectiva a altas tasas de muestreo es la interpolación lineal.

Una explicación muy detallada escapa a los alcances de esta nota de aplicación, por lo que simplemente enumeraremos las operaciones a realizar:

- ➔ Obtener una nueva muestra (A)
- ➔ Obtener la muestra demorada (B)
  - ➔ Interpolar dos muestras demoradas, 2 multiplicaciones de 16x16 + una suma de 16-bits
- ➔ Control de realimentación
  - ➔ Multiplicar por el coeficiente de realimentación, multiplicación de 16x16
  - ➔ Sumar a la nueva muestra, suma de 16-bits
- ➔ Guardar en el buffer circular
- ➔ Control de profundidad de modulación
  - ➔ Multiplicar A por el coeficiente de la señal sin procesar, multiplicación de 16x16
  - ➔ Multiplicar B por el coeficiente de la señal procesada, multiplicación de 16x16
  - ➔ Sumar A+B, 16-bits

- ➔ Entregar al DAC
- ➔ Control de ancho de modulación
  - ➔ Actualizar el LFO, suma de 32-bits
  - ➔ Revisar si excede los límites e invertir sentido, resta (comparación) de 16-bits
- ➔ Calcular el nuevo retardo, suma de 16-bits

Hardware

El hardware a utilizar corresponde al desarrollado en CAN-063 y utilizado en CAN-065.

Software

Marco de trabajo

Utilizamos el esquema desarrollado en CAN-065, con una pequeña modificación en la rutina de interrupción del ADC, dado que no necesitamos guardar directamente la muestra obtenida como en el FIR, y utilizaremos un registro adicional, el SFR B

```
void ADInterrupt(void) interrupt 2 __naked using 1
{

_asm
  setb P2.0
        push psw
        push dpl
        push dph
        push ACC
        push b
        mov psw,#0x8                ; registros: bank 1

        mov dph,_SPIRXTX1
        mov dpl,_SPIRXTX0
        anl dph,#0x0F               ; dp = adcdata, 12-bit offset binary
        mov a,dph                   ; adcdata <<= 4
        swap a                      ; copied from sdcc
        anl a,#0xf0
        xch a,dpl
        swap a
        xch a,dpl
        xrl a,dpl
        xch a,dpl
        anl a,#0xf0
        xch a,dpl
        xrl a,dpl                   ; 16-bit offset binary
        xrl a,#0x80                 ; convierte offset binary a 2s-complement en 1.15
        mov dph,a

        mov A,_Flangeron
        jz flgoff
  setb P2.1
        lcall _Flanger               ; adcdata=FIR(datastack,fircoef,FIRLEN);
  clr P2.1
flgoff:
        mov A,dph                    ; 8-bit
        xrl A,#0x80                  ; convierte a 8-bit offset binary
                                     ; if(adcdata==0)    // correct PWM flaw
       jnz no0
       inc A                         ; adcdata++

no0:    mov _PWMCFG,#0x05            ; PWM5 MID LSB
       mov _PWMDATA,A
        pop b
        pop ACC
        pop dph
        pop dpl
        pop psw
  clr P2.0
        reti
_endasm;
}
```

Flanger

Comenzaremos primero con una simple rutina que convierte los parámetros de operación de un formato humanamente comprensible a otro utilizable por el micro. Dicha función recibe los valores de tiempo en milisegundos, los controles en porcentaje, y la velocidad de modulación (frecuencia del LFO) en décimas de Hz (1=0,1Hz); completando todas las variables que necesita el algoritmo en su formato, es decir, los tiempos en muestras, los controles en coeficientes, y la velocidad de modulación en paso por muestra (cuanto se incrementa/decrementa en cada muestra).

```
unsigned int fp,ep;               // punteros
unsigned int max_sweep,min_sweep; // límites de barrido (offsets dentro del buffer)
unsigned int sweep_h,sweep_l,sweep_step;   // barrido acumulado (32 bits 16.16) y paso 0.16
int wet_coef,dry_coef,fbk_coef;    // coeficientes de suma 1.15
unsigned char sweep_dir=0;         // dirección del barrido

// Muestras por ms
#define SPMS 39

void setupFlanger(int delay,int depth,unsigned char wet,unsigned char dry,
               unsigned char fback,unsigned char step)
{
       sweep_step=13*step*depth;     // paso por muestra en 0.16: 2*depth*step/samp =
                                     // 65536*2*depth*(step/10)/(1000*SPMS)
       delay*=SPMS;                  // retardo en muestras
       depth*=SPMS;                  // ancho en muestras
       max_sweep=(BUFFERLEN-2)-delay;
       min_sweep=max_sweep-depth;
       sweep_h=min_sweep;
       sweep_l=0;
       wet_coef=327*wet;             // 1.15: 32768*wet%/100 (32768 no está permitido)
       dry_coef=327*dry;
       fbk_coef=327*fback;
       fp=0;                         // inicializa punteros
       ep=(min_sweep<<1);
}
```

A continuación desarrollamos la rutina del efecto, todo el procesamiento lo hacemos en 16-bits, con números normalizados en formato 1.15.

Para simplificar el manejo del buffer circular, lo alojamos en memoria externa al comienzo de la misma (posición 0x0000). Como trabajamos con datos en 16-bits, con la restricción de que el tamaño del buffer sea mayor a 256 posiciones (512 bytes) y una potencia de 2, sólo deberemos preocuparnos por el byte más significativo del puntero y realizar una operación de AND.

Para pasaje de parámetros entre la rutina de interrupción y la de Flanging utilizamos el método que emplea SDCC, el compilador utilizado, por generalidad. Lo mismo hacemos con el valor devuelto por esta rutina, que será escrito en el DAC vía PWM por la rutina anterior.

```
// debe ser potencia de 2, >= 256
#define BUFFERLEN 256
// máscara para el buffer circular, MSB (256-> 1, 512 -> 3, 1024 -> 7 ...)
#define BUFMASK 1


unsigned char Flangeron=1;

__xdata at 0x0000 static int databuf[BUFFERLEN];


int Flanger(int cursamp) __naked
{
cursamp=cursamp;
_asm

  orl _DEVMEMCFG,#0x01        ; SFR Page 1
  mov _AUCONFIG1,#0x08        ; CAPREV  = 0  captura automática
                                     ; CAPMODE = 0   al cargar AUA0
                                     ; OVCAPEN = 0  no captura en overflow
                                     ; READCAP = 0  AURES =result
                                     ; ADDSRC  = 10 Add SRC = AUPREV
                                     ; MULCMD  = 00 Mul cmd = AUA x AUB
```

```
  mov _AUCONFIG2,#0xA0        ; limpia registros
  mov _AUSHIFTCFG,#0x00       ; no usa barrel shifter

  mov r2,dpl            ; salva muestra actual
  mov r3,dph
  mov dpl,_ep           ; puntero a muestra demorada
  mov dph,_ep+1
  movx a,@dptr          ; la obtiene
  mov _AUA0,a           ; interpola out=next*frac+cur*(1-frac)
  inc dptr              ; no debo revisar circularidad hasta operar sobre una unidad
  movx a,@dptr
  mov _AUA1,a           ; AUA=cur
  clr C
  clr a
  subb a,_sweep_l       ; AUB=(1-frac) en 0.16, 1 es el bit 17, frac=0 no está permitido
  mov _AUB0,a
  clr a
  subb a,_sweep_l+1
  clr c                 ; convierte a 1.15, la MAC es con signo y esta operación no
  rrc a                 ; bit 16 será 0
  mov _AUB1,a
  mov a,_AUB0
  rrc a
  mov _AUB0,a
  orl a,_AUB1           ; frac=0 ? (1-frac será 0 si frac=0, en 0.16)
  jnz interp            ; No, sigamos
  mov _AUB0,#1          ; Sí, salteo interpolación.
                        ; (AUB1=0) AUB=1 en 16.0, entonces AURES=cur en 17.15
  sjmp nointerp
interp:
  inc dptr              ; muestra demorada siguiente
  anl dph,#BUFMASK      ; circularidad del buffer
  movx a,@dptr
  mov _AUA0,a           ; captura cur*(1-frac) en 2.30
  inc dptr
  movx a,@dptr
  mov _AUA1,a           ; AUA=next
  mov a,_sweep_l+1      ; AUB=frac en 1.15
  clr c
  rrc a
  mov _AUB1,a
  mov a,_sweep_l        ; next*frac en 2.30, acumula
  rrc a
  mov _AUB0,a
  mov _AUSHIFTCFG,#0x31 ; El barrel shifter desplaza 15-bits a la derecha (2.30 a 1.15)
nointerp:               ; cuidado, pasamos por acá con diferentes seteos del barrel shifter
  mov r4,_AURES0        ; guarda el valor de salida. Si interpolamos,
                        ; out=AUPREV+AUA*AUB=next*frac+cur*(1-frac) en R4 R5 en 1.15
  mov r5,_AURES1        ; si no interpolamos, out=cur en 17.15=1.15
  mov _AUA0,_fbk_coef   ; captura resultado en AUPREV
  mov _AUA1,_fbk_coef+1 ; carga coeficiente de realimentación en 1.15
  mov _AUCONFIG1,#0x02  ; AUA*AUPREV result
                                          ; ADDSRC  = 00 Add SRC = 0
                                          ; MULCMD  = 10 Mul cmd = AUA x AUPREV
  mov _AUSHIFTCFG,#0x31 ; El barrel shifter desplaza 15-bits a la derecha (2.30 a 1.15)
  mov dpl,_fp
  mov dph,_fp+1
  mov a,_AURES0         ; Control de realimentación: out*feedback en 1.15
  add a,r2              ; suma a la muestra actual
  movx @dptr,a          ; guarda in+fbk*buf en el buffer
  mov a,_AURES1
  addc a,r3
  jnb PSW.2,fbk_ok      ; si hay overflow, recorta; esto evita oscilaciones desagradables
  mov b,a
  mov a,#0xFF
  movx @dptr,a
  mov a,#0x7F
  jb b.7,fbk_ok
  clr a
  movx @dptr,a
  mov a,#0x80
fbk_ok:
  inc dptr
```

```
  movx @dptr,a          ; guarda MSB en el buffer
                        ; salva dptr para usarlo más adelante
  mov _AUCONFIG1,#0x08  ; AUPREV + AUA x AUB
                                          ; ADDSRC  = 10 Add SRC = AUPREV
                                          ; MULCMD  = 00 Mul cmd = AUA x AUB

  mov _AUCONFIG2,#0xA0        ; Limpia los registros de la MAC
  mov _AUSHIFTCFG,#0x00       ; No usa barrel shifter
  mov _AUA0,r2
  mov _AUA1,r3
  mov _AUB0,_dry_coef         ; Control de Profundidad de modulación
  mov _AUB1,_dry_coef+1        ; señal directa: input sample * dry level, 2.30
  mov _AUA0,r4
  mov _AUA1,r5
  mov _AUB0,_wet_coef
  mov _AUB1,_wet_coef+1        ; + señal procesada * wet level, 2.30

  mov _AUSHIFTCFG,#0x01        ; barrel shifter desplaza 1 bit a la izquierda (2.30 a 1.31)
                               ; tomaremos el valor en 1.15 más adelante
  mov a,_sweep_dir            ; procesa el LFO
  jnz godown
  mov a,_sweep_l               ; rampa 'ascendente', suma en 32-bits
  add a,_sweep_step
  mov _sweep_l,a
  mov a,_sweep_l+1
  addc a,_sweep_step+1
  mov _sweep_l+1,a
  mov a,_sweep_h
  addc a,#0
  mov _sweep_h,a
  mov a,_sweep_h+1
  addc a,#0
  mov _sweep_h+1,a
  sjmp incptrs

godown:
  clr c
  mov a,_sweep_l               ; rampa 'descendente', resta en 32-bits
  subb a,_sweep_step
  mov _sweep_l,a
  mov a,_sweep_l+1
  subb a,_sweep_step+1
  mov _sweep_l+1,a
  mov a,_sweep_h
  subb a,#0
  mov _sweep_h,a
  mov a,_sweep_h+1
  subb a,#0
  mov _sweep_h+1,a
incptrs:                       ; incrementa punteros
  mov a,_fp
  add a,_sweep_h               ; ep = (int *) fp + sweep_h
  mov _ep,a
  mov a,_fp+1
  addc a,_sweep_h+1
  mov _ep+1,a
  mov a,_ep
  add a,_sweep_h               ; suma dos veces, el buffer es de unidades de 16-bits
  mov _ep,a
  mov a,_ep+1
  addc a,_sweep_h+1
  anl a,#BUFMASK               ; mantiene circularidad
  mov _ep+1,a
  inc dptr                     ; fp++
  mov _fp,dpl
  anl dph,#BUFMASK
  mov _fp+1,dph

                               ; control de ancho de modulación
  clr c                        ; chequea rampa e invierte sentido
  mov a,_sweep_h               ; sweep >= max_sweep ?
  subb a,_max_sweep
  mov a,_sweep_h+1
  subb a,_max_sweep+1
```

```
  jc chlow
  mov _sweep_dir,#1            ; sí, para abajo
  sjmp done
chlow:
  clr c
  mov a,_sweep_h               ; no, sweep < min_sweep
  subb a,_min_sweep
  mov a,_sweep_h+1
  subb a,_min_sweep+1
  jnc done
  mov _sweep_dir,#0            ; sí, para arriba
done:

  mov dph,_AURES3              ; obtiene el resultado en 1.15 en dptr, descartando 16 LSbs
  mov dpl,_AURES2

  mov a,_AUCONFIG2
  jnb acc.0,out_ok             ; revisa si hubo overflow
  mov dpl,#0xFF                ; y satura
  mov dph,#0x7F
  mov a,_AURES3
  jb acc.7,out_ok
  mov dpl,#0
  mov dph,#0x80
out_ok:

  anl _DEVMEMCFG,#0xFE         ; SFR Page 0
  ret
_endasm;
}
```

Programa principal

Finalmente, el programa principal, que en este caso no hace absolutamente nada más que inicializar el sistema y esperar ocioso, dado que es un ejemplo.

```
void main (void) {

        PERIPHEN2 |= (1<<5);                 // Habilita Unidad Aritmética
        initAD();
        initPWM();
         P2&=~0x1F;                          // Outputs = 0
         P2PINCFG&=~0x1F;                    // 11100000 P2.0,1,2,3,4 = output
        setupFlanger(0,6,70,70,20,1);
        GENINTEN = 0x03;                     // Global interrupts

        while (1){
        }
}
```