Cika Electrónica

Nota de Aplicación: CAN-065
Título: **DSP con Ramtron VRS51L3074: filtro FIR**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	12/02/07	

En esta nota de aplicación veremos un caso práctico de filtrado de señales con los Ramtron VRS51L3074. Si bien el fabricante provee una excelente nota de aplicación al respecto, consideramos necesario desarrollar un marco más flexible. En la nota de aplicación CAN-063 vimos una forma de aprovechar los generadores de PWM para reproducir audio y utilizar el MCP3204 para obtenerlo, según lo desarrollado en CAN-061; lo utilizaremos para entrada y salida de la señal.

Introducción

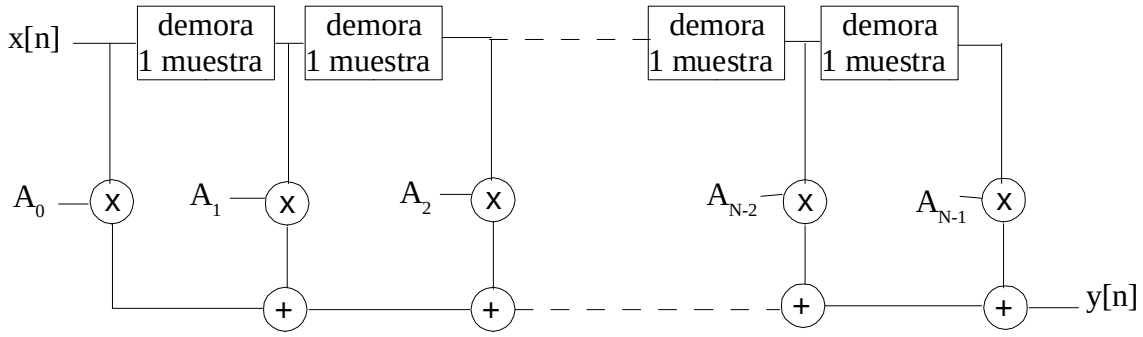
Si bien un filtro de tipo FIR de 16 etapas como el desarrollado requiere mucho de la CPU, gracias al esquema desarrollado en la CAN-063 podemos minimizar el tiempo ocioso y procesar en tiempo real una señal con un ancho de banda de poco más de 19KHz, quedándonos algo de tiempo para atender otras tareas

Filtros FIR

La función transferencia en el plano Z que caracteriza un sistema FIR (Finite Impulse Response) es la siguiente: $H[Z]=\sum_{k=0}^{N-1} A_k \cdot Z^{-k}$, donde A_k son coeficientes y N es la cantidad de taps del filtro.

La ecuación diferencia que implementa esta transferencia es la siguiente: $y[n]=\sum_{k=0}^{N-1} A_k \cdot x[n-k]$, donde *y[n]* es la secuencia de salida, *x[n]* la secuencia de entrada, A_k son coeficientes, *x[n-k]* corresponde al valor de la secuencia *x[n]* un instante *k* muestras anterior al momento actual, y *N* es la cantidad de taps del filtro, longitud de la respuesta al impulso en muestras.

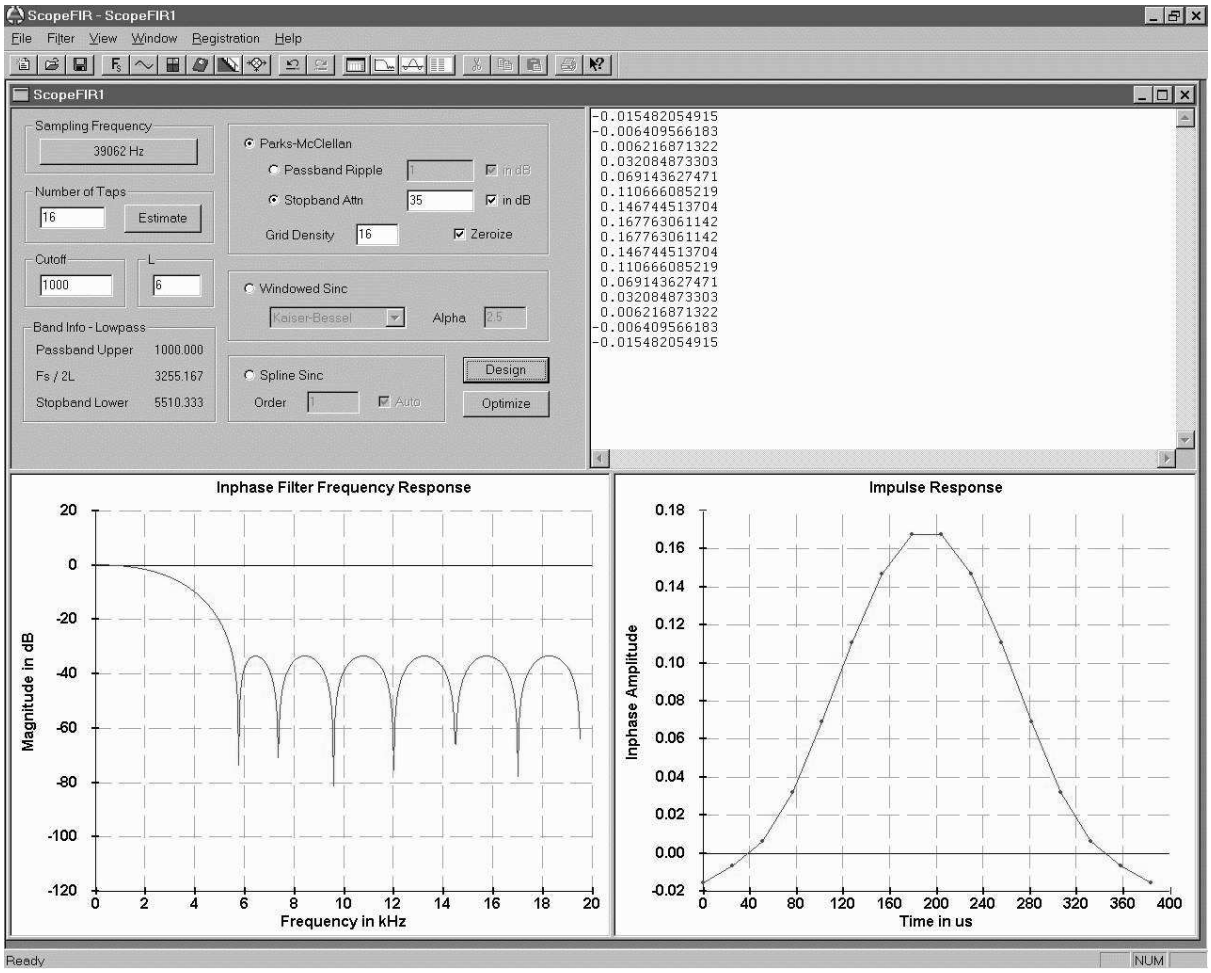
Si bien existen varias formas de implementarlo, la más simple y explícita es la siguiente:



Como puede observarse, esto sugiere la necesidad de emplear gran cantidad de sumas y multiplicaciones con números del formato que la precisión de la señal requiera, y realizarlo a gran velocidad, dado que deben realizarse todas estas operaciones antes de que la próxima muestra esté disponible. Así mismo, vemos también que una MAC (Multiply and ACcumulate) resulta de suma utilidad para realizar este algoritmo, dado que nos simplifica notoriamente la tarea de multiplicar (sólo debemos cargar un par de registros) y también la de acumular (no necesitamos propagar carries).

Los valores de los coeficientes se determinan de acuerdo a las necesidades de filtrado, utilizando diversos métodos de síntesis; en nuestro caso calculamos un filtro que comience a cortar en 1KHz y atenúe 35dB a

5KHz. Utilizamos una versión demo de un conocido programa shareware, y tomamos los valores de la pantalla:



Sistemas de numeración

Por lo general, los coeficientes son números reales de valor absoluto menor que la unidad, por lo que resultan fácilmente aproximados por números en coma fija en el formato 1.15, es decir, un dígito para signo y parte entera, y 15 dígitos para la parte fraccionaria.

La MAC presente en los VRS51L3074 y 2070 es entera. Al multiplicar dos números fraccionarios en formato 1.15 con una MAC entera, obtendremos el resultado en formato 2.30; pero como la multiplicación de dos números menores que la unidad no puede dar más que uno, lo que sucede es que se duplica el bit de signo. Mediante el uso del barrel shifter, es posible desplazar todo el resultado una posición a la izquierda y obtener 1.31, cuya palabra más significativa corresponde al resultado deseado en formato 1.15.

Hardware

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

Software

Marco de trabajo

Utilizamos el esquema desarrollado en CAN-063, por lo que no repetiremos esa parte aquí. Sí comentaremos algunas particularidades:

➢

- A fin de no tener que salvar demasiados registros, utilizaremos en las rutinas relacionadas con DSP el banco 1 de registros.

- Debido al tiempo que emplea la rutina FIR, se produce la interrupción del PWMtimer antes que ésta termine, efectivamente interrumpiendo al procesamiento, que fue iniciado por la interrupción de SPI. Para que esto sea posible, deberemos asignar mayor prioridad al módulo PWMtimer, lo que hacemos operando sobre el registro INTPRI2:

```
INTPRI2 |= (1<<5);                    // Eleva la prioridad de interrupción de PWM7:4
```

- Dado que el FIR utiliza la MAC y ésta está en la segunda página de SFRs, y esta rutina resulta a su vez interrumpida por la del PWMtimer que necesita utilizar la primera página de SFRs, debemos salvar el SFR DEVMEMCFG en esta rutina de interrupción, así como también re-seleccionar el banco de registros 0, lo cual realiza el compilador pero mostramos en assembler de todos modos. La operación sobre P2.2 es para poder observar el instante de la interrupción

```
void PWMTMRInterrupt(void) interrupt 13
{
        // sólo usamos un timer, no comprobamos cuál interrumpe
P2|=(1<<2);
        __asm
        push _DEVMEMCFG
        push psw
        mov psw,#0x00
        __endasm;
        DEVMEMCFG&=~(1<<0);
        PWMTMRF &= ~(1<<4);

       SPIRXTX2 = 0x00;
       SPIRXTX1 = (CHANNEL<<6);
       SPIRXTX0 = START | SINGLE | (CHANNEL>>2);
        __asm
        pop psw
        pop _DEVMEMCFG
        __endasm;
P2&=~(1<<2);
}
```

Hemos realizado además una modificación en la rutina de interrupción del ADC, para normalizar el valor obtenido. El ADC nos entrega un número en 12-bits offset binary, pero nuestras rutinas matemáticas trabajarán con números normalizados en formato 1.15, que es complemento a dos; entonces deberemos convertir este valor a una fracción normalizada, desplazándolo cuatro posiciones a la izquierda y sumándole 0x8000, lo que equivale a invertir el bit más significativo. A fin de determinar la cantidad de CPU que utiliza esta rutina, movemos el pin P2.0 al comienzo y al final, y el pin P2.1 al entrar y salir de la rutina de DSP.

El keyword *__naked* significa que el compilador no inserta código para salvar registros. El keyword *using 1* indica nuestra voluntad de utilizar el set 1 de registros, por lo que el compilador usará este set para esta rutina (si escribiéramos algo en C) y manejará en el resto el cambio de sets de registros.

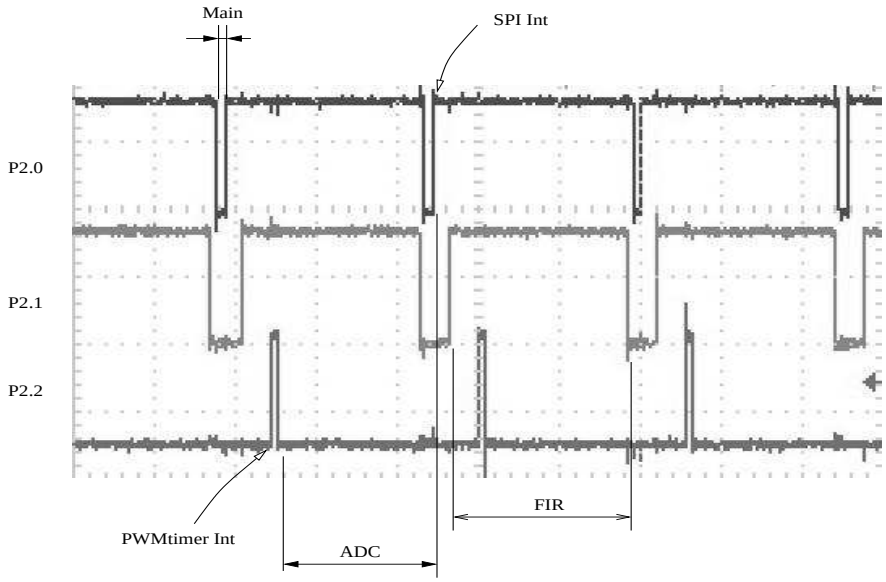
```
void ADInterrupt(void) interrupt 2 __naked using 1
{

_asm
  setb P2.0
        push psw
        push dpl
        push dph
        push ACC
        mov psw,#0x8                ; registros: bank 1
       // Toma muestra
        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 de offset binary a 2s-complement en 1.15
        mov dph,a

       mov R0,#_datastack           ; copia muestra en buffer
        mov @R0,dpl
        inc R0
        mov @R0,dph                 ; datastack[0] = adcdata;
        mov A,_FIRon
        jz firoff
  setb P2.1
        mov dpl,#_datastack          ; parámetros de llamada
        mov _FIR_PARM_2,#_fircoef
        mov _FIR_PARM_3,#FIRLEN
        lcall _FIR                   ; adcdata=FIR(datastack,fircoef,FIRLEN);
  clr P2.1
firoff:
        mov A,dph                    ; descarta LSB, 8-bit data
        xrl A,#0x80                  ; convierte a 8-bit offset binary
                                     ; if(adcdata==0)    // correct PWM flaw
       jnz no0
       inc A                         ; adcdata++
       // escribe muestra
no0:   mov _PWMCFG,#0x05             ; PWM5 MID LSB
       mov _PWMDATA,A
        pop ACC
        pop dph
        pop dpl
        pop psw
  clr P2.0
        reti
_endasm;
}
```

En el diagrama siguiente puede apreciarse la secuencia de interrupciones y el tiempo ocupado. Seguramente, optimizando un poco más los algoritmos y el código podríamos obtener un poco más de performance, sin embargo, para una nota de aplicación con un micro de 8-bits, procesar en 16-bits un FIR de 16 taps a 39062 muestras por segundo nos pareció más que suficiente.



FIR

A continuación desarrollamos la rutina de FIR propiamente dicha, para lo cual aprovechamos los ejemplos provistos por el fabricante. Necesitamos ir almacenando N-1 muestras previas, las cuales guardaremos en un buffer en RAM. A fin de que la misma rutina pueda utilizarse para diversas aplicaciones, mantenemos tanto la cantidad de taps como el buffer de datos y el array de coeficientes como parámetros. Los coeficientes los definimos en una variable inicializada, y el compilador se encargará de moverlos desde flash. Si necesitamos varios sets de coeficientes para diversos filtros, podemos definir varias variables y pasamos el puntero correspondiente.

El algoritmo del FIR en sí es como se observa en el diagrama de la primera página, y no difiere mucho del código provisto por Ramtron. Nuestra principal modificación aquí ha sido trabajar con signo y números normalizados 1.15. Tomamos entonces cada una de las muestras y la multiplicamos por su respectivo coeficiente. A fin de evitar el manejo de un buffer circular, particularmente cuando la cantidad de taps no es potencia de dos, trabajamos sobre posiciones fijas en el buffer y al terminar el procesamiento desplazamos todas las muestras un lugar.

Para pasaje de parámetros entre la rutina de interrupción y la de FIR 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.

```
// Coeficientes en 1.15
// FSAMPLE 39062HZ, N=16, LOW PASS 1KHZ -35DB @ 5KHZ

int fircoef[] = {
0xFE05,  //-0.0154821
0xFF2E,  //-0.0064096
0x00CC,  // 0.0062169
0x041B,  // 0.0320849
0x08DA,  // 0.0691436
0x0E2A,  // 0.1106661
0x12C8,  // 0.1467445
0x1579,  // 0.1677631
0x1579,  // 0.1677631
0x12C8,  // 0.1467445
0x0E2A,  // 0.1106661
0x08DA,  // 0.0691436
0x041B,  // 0.0320849
0x00CC,  // 0.0062169
0xFF2E,  //-0.0064096
0xFE05   //-0.0154821
};

__idata int datastack[FIRLEN];
unsigned char FIRon=1;

int FIR(__idata int *dat,__idata int *coef,unsigned char len) __naked
{
  dat=dat;                              // esto evita que el compilador nos diga que no
  coef=coef;                            // utilizamos los parámetros
  len=len;

        _asm

        orl _DEVMEMCFG,#0x01            ; SFR Page 1
        mov _AUCONFIG1,#0x08            ; CAPREV  = 0  captura automática
                                                ; CAPMODE = 0  cuando se escribe AUA0
                                                ; OVCAPEN = 0  no captura con overflow
                                                ; READCAP = 0  AURES =resultado
                                                ; 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 R0,_FIR_PARM_2              ; tabla de coeficientes del FIR
        MOV R1,dpl                      ; buffer de datos
        MOV R2,_FIR_PARM_3
  FIRloop:
               MOV  _AUA0,@R0                   ; copia LSB coeficiente
```

```
                INC  R0                         ; esto captura en AURES la última multiplicación
                MOV  _AUA1,@R0                  ; copia MSB coeficiente
                INC  R0
                MOV  _AUB0,@R1                  ; copia LSB dato
                INC  R1
                MOV  _AUB1,@R1                  ; copia MSB dato
                INC  R1                         ; mult/acc en un ciclo
                DJNZ R2,FIRloop                 ; acumula en 2.30

        MOV R2,_FIR_PARM_3              ; Taps del filtro
        DEC R2                          ; -1
        MOV A,R2
        ADD A,R2
        INC A
        ADD A,dpl                        ; MSB último tap
        MOV R1,A
        DEC A
        DEC A
        MOV R0,A                         ; MSB tap n-1
  FIRshift:
                mov  A,@R0                      ; mueve datos
                mov  @R1,A                      ; evita buffer circular
                dec  R0
                dec  R1
                mov  A,@R0
                mov  @R1,A
                dec  R0
                dec  R1
                DJNZ R2,FIRshift

         mov _AUSHIFTCFG,#0x01           ; desplaza una posición a la izquierda
                                         ; obtiene resultado en 1.31
         mov dph,_AURES3                 ; toma 1.15 descartando 16 LSbs
         mov dpl,_AURES2

        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
        GENINTEN = 0x03;                        // Global interrupts

        while (1){
        }
}
```