Cika Electrónica

Nota de Aplicación: CAN-023
Título: **Osciloscopio de almacenamiento con PG320240FRST, ADS7846 y Dynamic C 8**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	20/09/04	

Basándonos en hardware y software ya desarrollados, utilizamos las entradas libres del controlador ADS7846 para emplearlo como conversor AD y poder hacer una suerte de osciloscopio de almacenamiento (storage scope), principalmente orientado a procesos, dada su baja velocidad de muestreo y carencia de circuito de disparo.

Esta nota de aplicación hace uso de software y hardware desarrollado en las siguientes notas de aplicación:
CAN-016: portamos las bibliotecas de funciones con soporte para displays gráficos y pantallas sensibles al tacto (touch screens), que incluye Dynamic C en su versión 8, para su utilización con el display PG320240FRST de Powertip.
CAN-020: hardware de lectura de touch screen con ADS7846.
CAN-021: agregado de funciones de soporte del controlador ADS7846 en nuestra biblioteca de funciones: *Cika320240FRST.lib.*
A su vez, estas notas de aplicación pueden hacer referencia a otras, las cuales abordarán en mayor profundidad algunos temas como estructura y drivers del display gráfico y/o la pantalla sensible al tacto. Se recomienda la lectura de las notas de aplicación mencionadas y sus referencias.

Descripción del proyecto

Partimos de la premisa de aprovechar el hecho de disponer de entradas auxiliares en el controlador de touch screen empleado, lo que nos permite utilizarlo como un conversor analógico digital. Dado que se trata de un conversor serie, y que lo estamos compartiendo con todo el hardware y software de lectura de touch screen, no nos es posible realizar muchas lecturas a gran velocidad. Por estos motivos, nos orientaremos más a lo que se suele denominar “process scope”, es decir, un osciloscopio destinado a mostrar variables de proceso, como por ejemplo la señal analógica proveniente de un medidor de caudal, o alguna otra similar de relativamente baja frecuencia de variación. A tal fin, podemos prescindir del sistema de trigger, dado que más que una forma de onda, nos interesa observar las variaciones de una señal de contínua. Queda como inquietud para el lector el desarrollo de dicha parte del instrumento, si es de su interés.

Como se trata de un oscilocopio de almacenamiento, tomaremos una cierta cantidad de muestras y las mostraremos en pantalla; cuando la velocidad de muestreo sea lo suficientemente lenta, preferiremos mostrar la señal a medida que se la va muestreando. No debemos olvidarnos de que se trata de un muestreo, y que la velocidad de variación de la señal deberá respetar los criterios de Nyquist, y el famoso teorema del muestreo, es decir, el ancho de banda de la señal a medir deberá ser inferior a la mitad de la frecuencia de muestreo.

Haciendo uso y abuso del código provisto por Rabbit en forma de bibliotecas de funciones (libraries), agregaremos una opción para mostrar el espectro de la señal, llamando a funciones de transformada rápida de Fourier de la biblioteca *FFT.lib.*

El control del instrumento se realizará mediante un juego de simples botones, que aumentan o disminuyen la ganancia (amplificación vertical) y la frecuencia de muestreo. A los fines de mantener el proyecto simple, y dado que la resolución del conversor AD utilizado (12-bits=4096 valores) es mucho mayor que la de la pantalla (224 puntos, apróx 8-bits), simplemente multiplicaremos las muestras por una constante, evitando desarrollar y tener que controlar un amplificador analógico. Si bien esto no es óptimo, cumple los requerimientos de una nota de aplicación.

Por supuesto que este proyecto no es en sí de gran utilidad, pero a los efectos de lo que en realidad es, una nota de aplicación, nos presenta una alternativa interesante para un sistema posible, planteando la forma de resolver algunos interrogantes que surgen durante el desarrollo del mismo. Un sistema profesional emplearía un conversor paralelo rápido leído por interrupciones a tiempo constante, y tal vez hasta un buffer por

hardware para lograr mayor velocidad, pero para los alcances e intención de esta nota de aplicación, creemos suficientemente útiles las características y arquitectura presentadas

Hardware

El hardware es el mismo utilizado en la CAN-021, desarrollado en la CAN-020; simplemente utilizaremos los pines adicionales: AUX, V_{BAT} y V_{REF} para funcionar como entradas de señal y referencia. El pin AUX debería tener alguna clase de protección, de modo tal que la tensión a medir no exceda los límites recomendados de operación del conversor. El pin V_{BAT} es algo más robusto, dado que está orientado a medir una batería de valor más elevado y posee ya una resistencia serie, pero no está de más protegerlo de todos modos. El pin V_{REF} es la referencia contra la cual se medirá plena escala, no debe superar el valor de la tensión de alimentación, y para los fines prácticos lo conectaremos a la misma (Vcc). Mediremos entonces la entrada AUX en modo single-ended, utilizando V_{REF} como referencia (conectado a Vcc), que nos dará 5V a plena escala.

Desarrollo del driver

Como vimos en la CAN-014, y luego en la CAN-021, la función *_adcTouchScreen()* es la que realiza la lectura del chip controlador de la touch screen. El parámetro que se le pasa es el que indica qué entrada leerá. Simplemente re-utilizamos el código ya existente, pasándole a esta función un parámetro tal que haga que el controlador devuelva el valor presente en la entrada AUX, en 12-bits (un valor entre 0 y 4095).

Lo único que tenemos que tener en consideración es que el software encargado de tomar muestras es mutuamente excluyente con el de lectura de la touchscreen, en el sentido que ambos comparten al conversor. Esto nos impide el leer el conversor por interrupciones, a menos que desarrollemos un complicado sistema de semáforos para compartir el recurso.

Software

Desarrollaremos las rutinas de soporte para sampleo y presentación, y luego el programa principal, que es el que se encarga de manejar los botones en pantalla. El display utilizado tiene una resolución de 320x240 puntos, lo cual nos permite emplear 256 puntos en sentido horizontal y 224 en sentido vertical como pantalla del instrumento, dejando el área restante como "frente" del mismo, con los correspondientes controles. La pantalla del instrumento se extenderá entonces de las posiciones 1 a 256 inclusive, en sentido horizontal, y 8 a 231 inclusive, en sentido vertical. Estos valores son los responsables de los caprichosos números que modifican y escalan las muestras para formar la imagen.

A los fines prácticos, y para mantener la claridad (de eso se trata una nota de aplicación), utilizaremos un costate para controlar el dibujo de los botones en pantalla, otro para atender a los botones, y otro para el instrumento en sí (muestreo y presentación). Este último, hace uso de cofunciones, lo que nos permite dar, a su vez, un ejemplo de uso de las mismas, manteniendo mayor claridad en el programa principal. Si bien la lectura de touchscreen (manejo de botones) y el muestreo de la señal están en costates separados, debido a que se trata de multitarea cooperativo, cada uno cede el control al otro cuando no está desarrollando ninguna tarea, y si el lector se toma el trabajo de observar el código en la biblioteca de funciones, la rutina de lectura del conversor no devuelve el control, por lo que no existe posibilidad de que un costate "moleste" al otro, perturbando el ciclo de lectura del conversor. Lo único que, a las velocidades de muestreo más altas, la "respuesta" de la pantalla puede resultar algo lenta.

Por supuesto que un osciloscopio profesional debería realizar una interpolación del tipo $\frac{\sin(x)}{x}$ o algún algoritmo propietario antes de mostrar la señal en pantalla; a los fines prácticos de esta nota de aplicación, mostraremos siempre un número fijo de muestras en pantalla, correspondiente al ancho de la misma, y utilizaremos interpolación lineal entre las muestras, uniendo con una recta los distintos puntos correspondientes a las muestras de la función a desplegar en pantalla. Si la señal es de frecuencia bastante menor (unas 10 veces o más) que la velocidad de muestreo, la interpolación lineal es óptima; si se acerca al límite de Nyquist, los puntos estarán lo suficientemente cerca como para enmascarar los errores de interpolación. Como no se realiza filtrado alguno, podrá presentarse aliasing si el usuario intenta hacer subsampling, es decir, tomar muestras de la señal a menor velocidad que la permitida por el teorema del muestreo.

Globales

Primero definiremos las variables globales, como el área de botones y el buffer para las muestras. También definiremos algunas constantes para la cantidad de muestras a tomar y la cantidad de escalas del instrumento. Incorporamos una variable global, *redraw*, que nos permitirá abortar el despliegue en pantalla y redibujar los botones cuando se modifica algún parámetro de operación del instrumento.

```
#memmap xmem
#class auto
// Usamos el hardware de lectura de la touch screen basado en ADS7846
#define TSCONTROLLER  2
/* Esto incluye la biblioteca de funciones de Cika que soporta el display
de 320x240 de Powertip */
#use "Cika320240FRST.lib"
/* Esta biblioteca provee funciones elementales que permiten utilizar
funciones más complejas en las demostraciones sin modificar las
bibliotecas originales de Rabbit */
//#use "CikaDemoKeybuzz.lib"

#use FFT.lib

// Definir para pasar las muestras a complemento a 2 (FFT)
//#define ACSCOPE

// cantidad de muestras
#define SAMPLES 256
// log2(SAMPLES/2), usado para escalar la FFT
#define L2_SAMPLES2   7

#define XSCALES 7
#define YSCALES 5
// comando a enviar al controlador para leer la entrada deseada
#define INPUT  0xE4

fontInfo fi8x10,fi10x12,fi6x8;

static unsigned int buffer[SAMPLES+2];              // buffer para las muestras

unsigned  long userX;                               // área de memoria para los botones
int redraw;
```

Muestreo y presentación en pantalla (modo osciloscopio)

Debido a que se trata de un conversor serie, y que lo estamos compartiendo con todo el hardware y software de lectura de touch screen, no nos es posible realizar muchas lecturas a gran velocidad. Tomaremos entonces una cierta cantidad de muestras y las mostraremos en pantalla, pero cuando la velocidad de muestreo sea lo suficientemente lenta, mostraremos la señal a medida que se la va muestreando. En este último caso, para distinguir el "barrido" actual de la imagen anterior, tomamos prestado el funcionamiento de algunos osciloscopios de afamadas marcas, que borran una cierta cantidad de pixels por delante del barrido. Dado que el software de control del display trabaja copiando periódicamente un buffer de memoria al display, y esto requiere tiempo, desactivamos temporariamente este proceso mientras estamos actualizando la pantalla.

Como tomamos $N = 256$ muestras y cada una corresponde a un punto en pantalla, cada punto mostrará el valor de una muestra tomada a la velocidad de muestreo, es decir, muestras cada $\frac{1}{f_m}$, dando un total de $\frac{N}{f_m} = \frac{256}{f_m}$ para el ancho de pantalla. Ej.:

si tomamos 100 muestras por segundo (100sps), tendremos un puntito cada $\frac{1}{100\,sps} = 10\,ms$, y el ancho de la pantalla corresponderá a $256 \times 10\,ms = 2{,}56\,seg$. Esto resulta similar a un osciloscopio convencional.

Veamos el software. Se trata de una cofunción a la que llamaremos con parámetros tales como espaciado entre muestras y ganancia vertical. La función tomará las muestras, devolviendo el control del procesador a las demás tareas mientras espera el tiempo necesario entre muestras, abortando si se produce un accionamiento del usuario sobre algún botón, y dibujando en pantalla la señal.

```
cofunc sampledraw(int delta, float scale)
{
unsigned int k,y,lasty,aux;

        if(delta>=100){                              // velocidades de muestreo lentas
            buffer[0]=_adcTouchScreen(INPUT);               // toma primera muestra
            aux=(unsigned int)(((long)scale*buffer[0])>>12); // escala
            lasty=(aux<225)? 231-aux:8;                     // “satura” si excede pantalla;
            glPlotDot(1,lasty);                             // dibuja en pantalla
            for(k=1;k<SAMPLES;k++){
                waitfor(DelayMs(delta));                    // espera hasta próxima muestra
                if(redraw==2){                              // aborta si es necesario
                   redraw=0;
                   abort;
                }
                glBuffLock();                            // impide actualización de pantalla
                glSetBrushType(PIXWHITE);
                glBlock(k,8,(k<(SAMPLES-10))? 10:(SAMPLES-k),224);  // simula barrido (borra)
                glSetBrushType(PIXBLACK);
                buffer[k]=_adcTouchScreen(INPUT);           // toma muestra siguiente
                aux=(unsigned int)(((long)scale*buffer[k])>>12); // escala
                y=(aux<225)? 231-aux:8;                     // ajusta
                glPlotLine(k,lasty,k+1,y);                  // dibuja e interpola
                glBuffUnlock();                             // actualiza pantalla
                lasty=y;
            }
        }
        else {                                       // velocidades de muestreo rápidas
            for(k=0;k<SAMPLES;k++){
                waitfor(DelayMs(delta));                    // espera el tiempo necesario
                if(redraw==2){                              // aborta si debe hacerlo
                  redraw=0;
                  abort;
                }
                buffer[k]=_adcTouchScreen(INPUT);           // toma muestra siguiente
            }
            draw(scale);                                    // dibuja buffer en pantalla
        }
        return(1);
}
```

La función *draw()* utilizada, es una versión reducida del algoritmo empleado para muestreo lento, que borra toda el área de pantalla del instrumento antes de dibujar, y la reutilizaremos más adelante

```
void draw(float scale)
{
unsigned int i,y,lasty,aux;

        glBuffLock();                                       // impide actualización
        glSetBrushType(PIXWHITE);
        glBlock(1,8,256,224);                               // borra pantalla instrumento
        glSetBrushType(PIXBLACK);
        aux=(unsigned int)(((long)scale*buffer[0])>>12);    // escala primera muestra
        lasty=(aux<225)? 231-aux:8;                         // ajusta
        glPlotDot(1,y);                                     // dibuja
        for(i=1;i<SAMPLES;i++){                             // procesa muestras restantes
            aux=(unsigned int)(((long)scale*buffer[i])>>12);
            y=(aux<225)? 231-aux:8;
            glPlotLine(i,lasty,i+1,y);                      // interpola
            lasty=y;
        }
        glBuffUnlock();                                     // actualiza pantalla
}
```

Muestreo y presentación en pantalla (modo espectro)

En cuanto a la FFT, utilizamos el algoritmo de transformada real que provee Rabbit. Este algoritmo empaqueta un número N de muestras (valores reales) como un array de $\frac{N}{2}$ valores complejos y utiliza un

algoritmo de FFT compleja de $\frac{N}{2}$ puntos, devolviendo un array de $\frac{N}{2}$ valores complejos correspondientes a $\frac{N}{2}$ muestras de la transformada de Fourier, conocida como DFT (Discrete Fourier Transform). Como nos interesa mostrar el espectro de amplitud de la señal, tomaremos los módulos de estos $\frac{N}{2}$ valores complejos y los repetiremos de modo de formar un array de N valores, que mostraremos en pantalla.

Sin entrar demasiado en detalles teóricos de análisis de espacios de variables continuas y discretas, podemos determinar de forma práctica que: si a una dada velocidad de muestreo f_m , tenemos una ventana de N muestras correspondiente a un período $T=\frac{N}{f_m}$, y obtenemos $\frac{N}{2}$ muestras de la transformada de Fourier, cada muestra k corresponderá a una componente de frecuencia $\frac{k \cdot f_m}{N}$, de modo que la muestra $k=\frac{N}{2}$ resulta $f=\frac{f_m}{2}$, el límite de Nyquist. Como duplicamos el número de puntos a N , para que ocupen todo el ancho de pantalla, tendremos entonces que el punto $2 \cdot k$ corresponderá a una componente de frecuencia $\frac{k \cdot f_m}{N}, \forall 0 \leqslant k < \frac{N}{2}$.

El proceso de empaquetado y obtención de la FFT empleado tiene algunas particularidades, cuyo significado científico el lector puede profundizar en textos de DSP; sepamos que el primer valor del array (parte real del valor de contínua) es el doble de lo esperado, y el valor siguiente (parte imaginaria) correspondería al valor de la muestra $\frac{N}{2}$. Además, los números resultan $\frac{N}{2}$ veces mayor a lo esperado. A los fines prácticos, dividimos el primer valor a la mitad y anulamos el segundo, de modo tal que al calcular el módulo lo dividamos por $\frac{N}{2}$, obteniendo un gráfico tal que, si la señal de entrada es $DC + A \cdot \sin(2 \pi f)$, veamos en pantalla los valores esperados (escalados) de DC y A para los puntos 0 y $2 \cdot \frac{N \cdot f}{f_m}$, respectivamente. Por supuesto que el lector interiorizado en DSP advertirá inmediatamente que dado que la FFT devuelve la transformada de una señal hecha periódica repitiendo infinitamente en ambos sentidos del tiempo los valores contenidos dentro de la ventana, resulta la señal implícitamente multiplicada por una ventana rectangular y su espectro convolucionado con el de esta última, produciendo un ensanchamiento de los picos (spectral leakage) y la aparición de colas, provenientes de los lóbulos laterales de la función espectral de la ventana cuadrada. Sin embargo, la utilización de una ventana nos introduciría errores en la medición del valor de contínua, lo cual es fundamental si orientamos nuestro instrumento a monitoreo de variables de proceso. Queda para el lector, si así lo desea, modificar el proyecto introduciendo la opción de multiplicar por una ventana de Hamming, por ejemplo, previo a realizar la transformada. Dejamos además, un fragmento de código en compilado condicional para convertir las muestras del conversor de offset binary a complemento a 2.

Volviendo al software, tomaremos una cierta cantidad de muestras, las procesaremos, y las mostraremos en pantalla. Utilizamos una cofunción a la que llamaremos con parámetros tales como espaciado entre muestras y ganancia vertical. La cofunción tomará las muestras, devolviendo el control del procesador a las demás tareas mientras espera el tiempo necesario entre muestras, abortando si se produce un accionamiento del usuario sobre algún botón, y finalmente procesando y dibujando en pantalla la señal. La FFT la realizamos usando una función de *FFT.lib*, que utiliza una variable llamada *blockexp* para conservar la escala usando menos bits, se trata de un exponente binario a manera de formato block floating point. La división por el número de muestras la realizamos fácilmente restando a este exponente el logaritmo en base 2 del número de muestras de la FFT, y para elevar los valores al cuadrado, al calcular el módulo, deberemos duplicar el valor de este exponente.

```
cofunc sampledrawfft(int delta, float scale)
{
int *bufferi,blockexp,i;
float power;

        for(i=0;i<SAMPLES;i++){
```

```
            waitfor(DelayMs(delta));
            if(redraw==2){
                redraw=0;
                abort;
            }
            buffer[i]=_adcTouchScreen(INPUT);
        }
        bufferi=(int *) buffer;                         // convierte a entero con signo (cabe)
        blockexp=0;
#ifdef ACSCOPE
        for(i=0;i<SAMPLES;i++)                          // convierte a ca2 (no usado)
             bufferi[i]-=2048;
#endif
        fftreal(bufferi,SAMPLES/2,&blockexp);           // obtiene FFT
        bufferi[0] /= 2;                                // normaliza DC
        bufferi[1] = 0;                                 // descarta fmax

        blockexp-=L2_SAMPLES2;                          // divide por SAMPLES/2
        blockexp*=2;                                    // factor de escala elevado al cuadrado
        power=pow(2,blockexp);                          // factor de escala al salir del formato
        for(i=0;i<SAMPLES/2;i++){            // calcula módulos: ||=sqrt(Re^2+Im^2)
            buffer[2*i]=(unsigned int) (sqrt((((float)bufferi[2*i]*bufferi[2*i])+((float)
bufferi[2*i+1]*bufferi[2*i+1]))*power)+0.5);
             buffer[2*i+1]=buffer[2*i];                 // copia al punto adyacente
        }
        draw(scale);                                    // dibuja en pantalla

}
```

Programa principal

Ingresamos ahora en el mundo del programa principal, el cual resulta ser bastante sencillo, ocupándose de los botones y de disparar el muestreo y display correspondiente al modo de operación, el cual puede seleccionarse mediante un botón.

```
void main()
{
int btn,level,xscale,yscale;
const static float scale[]={224,560,1120,2240,5600};
const static int delta[]={1000,500,200,100,50,20,10};
const static char *scales[]={"5V fs   ","2V fs   ","1V fs   ","500mV fs","200mV fs"};
const static char *deltas[]={"1 sps  ","2 sps  ","5 sps  ","10 sps ","20 sps ","50 sps ","100
sps"};

   redraw=level=1;
   yscale=0;
   xscale=XSCALES-1;

   brdInit();                                           // inicializa hardware
   glInit();                                            // inicializa lib gráfica

   glXFontInit ( &fi8x10,8,10,0x20,0x7E,Font8x10 );     // define fonts
   glXFontInit ( &fi10x12,10,12,0x20,0x7E,Terminal9 );
   glXFontInit ( &fi6x8,6,8,0x20,0x7E,Font6x8 );

   glBlankScreen();                                     // borra pantalla

   userX = btnInit( 7 );                                // define botones
   btnCreateText(userX,1,276,120,30,20,1,1,&fi8x10,"V+");   // nivel 1 = siempre
   btnCreateText(userX,2,276,141,30,20,1,1,&fi8x10,"V-");
   btnCreateText(userX,3,260,170,29,20,1,1,&fi8x10,"H-");
   btnCreateText(userX,4,290,170,29,20,1,1,&fi8x10,"H+");
   btnCreateText(userX,5,260,202,59,34,1,2,&fi10x12,"FFT"); // nivel 2 = sólo en osc
   btnCreateText(userX,6,260,202,59,34,1,3,&fi10x12,"Osc"); // nivel 3 = sólo en espectro
   btnAttributes(userX,1,0,0,0,0);
   btnAttributes(userX,2,0,0,0,0);
   btnAttributes(userX,3,0,0,0,0);
   btnAttributes(userX,4,0,0,0,0);
   btnAttributes(userX,5,0,0,0,0);
   btnAttributes(userX,6,0,0,0,0);
```

```
   while (1) {
        costate {                                           // panel de control
            if(redraw==1){                                  // redibujado al cambiar de modo
                glBlankScreen();
                while (!btnDisplayLevel(userX,1));          // botones nivel 1
                if(level==1){
                    while (!btnClearLevel(userX,3));        // borra botones nivel 3
                    while (!btnDisplayLevel(userX,2));      // botones nivel 2
                    glPrintf(280,0,&fi10x12,"OSC");
                }
                else {
                    while (!btnClearLevel(userX,2));        // borra botones nivel 2
                    while (!btnDisplayLevel(userX,3));      // botones nivel 3
                    glPrintf(280,0,&fi10x12,"FFT");
                }
                glPlotPolygon(4,0,7,257,7,257,232,0,232);   // dibuja pantalla instrumento
                glPrintf(270,20,&fi6x8,scales[yscale]);     // muestra escala vertical
                glPrintf(270,40,&fi6x8,deltas[xscale]);     // muestra sampling rate
                redraw=2;                                   // señaliza inst debe abortar
            }
        }
        costate {                                           // maneja botones
            waitfor (( btn = btnGet(userX)) >= 0 );         // espera botón
            switch (btn){                                   // procesa
                case 1:                                     // escala vertical
                    if(++yscale==YSCALES)
                        yscale=YSCALES-1;
                    redraw=2;                               // aborta dibujo si cambia escala
                    break;
                case 2:
                    if(--yscale==-1)
                        yscale=0;
                    redraw=2;
                    break;
                case 3:                                     // sampling rate
                    if(--xscale==-1)
                        xscale=0;
                    redraw=2;
                    break;
                case 4:
                    if(++xscale==XSCALES)
                        xscale=XSCALES-1;
                    redraw=2;
                    break;
                case 5:                                     // selecciona espectro
                    level=2;
                    redraw=1;                               // debe redibujar frente
                    break;
                case 6:                                     // selecciona osc
                    level=1;
                    redraw=1;
                    break;
            }
            glPrintf(270,20,&fi6x8,scales[yscale]);         // actualiza escalas
            glPrintf(270,40,&fi6x8,deltas[xscale]);
        }
        costate {                                           // instrumento
             if(level==1){                                        // osciloscopio
                 wfd sampledraw(delta[xscale],scale[yscale]);
             }
             else {                                               // espectro
                 wfd sampledrawfft(delta[xscale],scale[yscale]);
            }
        }
    }
}
```

Y finalmente, una función ficticia para poder compilar la biblioteca de funciones (ver CAN anteriores)

```
int buzzer()
{
}
```