Nota de Aplicación: CAN-029
Título: **Utilización de displays colorOLED (SSD1332) con Rabbit**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	08/06/05	
1	09/06/06	detalle en rutina screen dump

Les presentamos los nuevos displays gráficos color con tecnología OLED. En este caso, nos referimos al UG9664GFD, display de 96x64 pixels, basado en chips controladores SSD1332. Analizaremos más tarde el software de control y un simple programa demostración.

Hardware

El SSD1332 presenta una interfaz con tres modos posibles de trabajo: tipo Motorola (E, R/W, D/C), tipo Intel (RD, WR, D/C), y serie. El usuario tiene la libertad de elegir cual utilizar, y nosotros hemos elegido la modalidad Intel, que corresponde al funcionamiento del bus de Rabbit, y nos permite tener máxima velocidad de transferencia de datos. Como estos displays funcionan a 3V, utilizaremos la familia 3000 de Rabbit

Para la interfaz con el micro no es necesario ningún tipo de glue-logic, hacemos una conexión directa entre el bus del Rabbit y el LCD, como puede apreciarse en la tabla a la derecha. En la mayoría de los módulos de la familia 3000, deberemos utilizar el bus auxiliar de I/O, el cual utiliza al port A para los datos y parte del port B para las direcciones, como es este caso, en el que utilizamos un RCM3720.

Rabbit	OLED
PA.0	D0
PA.1	D1
PA.2	D2
PA.3	D3
PA.4	D4
PA.5	D5
PA.6	D6
PA.7	D7
IORD	RD
IOWR	WR
PB.2	D/C
PE.4	CS
RST	RST

Las señales RD y WR se toman de las correspondientes para operaciones en el espacio de I/O: IORD y IOWR. La señal CS es generada utilizando la facilidad de IOSTROBE que provee Rabbit. Se asigna el espacio de I/O utilizado, se configura el pin, y éste funciona como CS para ese rango de direcciones. Elegimos utilizar el port PE.4, el cual responde al rango de direcciones 0x8000 a 0x9FFF, dentro de este rango, las direcciones pares acceden a un registro del controlador del display (A0=0) y las impares al otro (A0=1).

Este tipo de displays permite regular el brillo y contraste mediante la modulación de la corriente de los OLEDs, la cual se toma de una fuente de tensión algo más alta, por lo general de unos 7 a 11V.

El display dispone, además, de un pin de reset, el cual podemos controlar a voluntad o conectar al reset del circuito. Para el desarrollo de esta nota de aplicación, simplemente lo conectamos al reset del módulo Rabbit.

Software

Breve descripción del display gráfico

Estos displays son sumamente versátiles, la memoria puede alojar una pantalla completa a una resolución de color de 16 bits por pixel (16bpp), y el controlador tiene una serie de primitivas gráficas que permiten dibujar líneas y rectángulos con o sin relleno, con un simple comando. Esto simplifica enormemente la tarea del procesador para la presentación de menúes y demás tareas de una interfaz con el usuario, haciéndolos atractivos incluso para procesadores más chicos o con menos memoria. No poseen generador de caracteres, pero es posible definir el área de memoria en la cual se escribe, lo que simplifica enormemente la tarea de dibujar las letras manualmente, permitiéndonos disponer de varios sets de caracteres.

La estructura de memoria de la pantalla es lineal, los pixels se agrupan horizontalmente, correspondiendo el primer byte de memoria al primer pixel de la primera línea de arriba a la izquierda, y el último byte a los últimos ocho pixels de la última línea de abajo a la derecha, para el modo de 8bpp. En el modo de 16bpp, deberemos reemplazar byte por word en el texto anterior. De todos modos, la estructura puede ser cambiada

alterando los registros del controlador que (valga la redundancia) la controlan. Por comodidad y similitud con otros displays ya estudiados, la mantendremos así, seteando el modo *remapped* del controlador.

El direccionamiento del byte a leer o escribir en memoria se hace mediante comandos, especificando la región de pantalla que nos ocupa, en pixels. Tiene además un contador autoincrementado, el cual apunta a la dirección siguiente luego de una lectura o escritura. Esto resulta óptimo para enviar los datos byte por byte hasta completar el área elegida; sea para dibujar un caracter o un ícono.

Una característica interesante del display, es que puede funcionar a una alta velocidad de acceso, sin necesidad de contención ni chequeo de flag de ocupado (busy).

Algoritmos

Si bien el display tiene muchas formas de utilización, en esta nota de aplicación desarrollaremos algoritmos en base a las más simples.

Como el display ya incorpora primitivas para dibujo de líneas y rectángulos, simplemente utilizaremos estos comandos. Para direccionar un punto, lo que haremos es trazar una línea de un punto de longitud.

Para graficar funciones, debemos tener en cuenta que la coordenada (0;0) se halla en el extremo superior izquierdo de la pantalla.

Para mostrar íconos, caracteres, o pantallas, deberemos definir el área de pantalla igual a la del bitmap en cuestión y enviar los datos, de esta forma se aprovechan los contadores autoincrementados y la estructura de memoria; esto se reduce simplemente a enviar todos los bytes corridos.

En el caso de pantallas, si comparamos la estructura de memoria del display con la forma de guardar imágenes de 24 bits en formato BMP, veríamos que son muy similares, por ejemplo: BMP va de abajo a arriba y el display de arriba a abajo, por lo que la imagen se ve espejada verticalmente; BMP usa tres bytes (B, G, R) para la información de color y el display utiliza dos bytes o uno, según el formato. Además, BMP incluye un encabezado de 54 bytes.

Por consiguiente, para adaptar una imagen, debemos llevarla a la resolución deseada, espejarla verticalmente, salvarla en formato BMP y luego de descartar los 54 bytes del comienzo, procesar la paleta al formato del SSD1332 en el modo de color que queramos utilizar.

Para imprimir textos, indicamos la posición en pantalla mediante sus coordenadas y restringimos el área de escritura al tamaño del caracter. Luego, escribiremos uno o dos bytes (según el modo de color) por cada pixel del caracter.

Desarrollo

Desarrollamos a continuación el software de base para manejo del display. Para una mejor comprensión, dada la complejidad del SSD1332, se recomienda consultar su hoja de datos.

Dada la gran cantidad de información a mover hacia el display para la presentación de pantallas, decidimos escribir la rutina básica de escritura en assembler. De igual modo, para maximizar la performance en el volcado de pantallas (screen dump), escribimos también una corta rutina en assembler que toma los datos directamente de *xmem* y los envía al display, en bloques. Dado que esta rutina manipula directamente el registro XPC, deberá hallarse en área root.

```
#define PORTA_AUX_IO

/* Low level functions */

#asm root
;@sp+2= dato/comando a escribir
;
Write_Com::
        ld hl,(sp+2)                        ; obtiene comando (LSB)
        ld a,l
        ioe ld (0x8000),a                   ; lo escribe
        ret

Write_Data::
        ld hl,(sp+2)                        ; obtiene dato (LSB)
        ld a,l
        ioe ld (0x8001),a                   ; lo escribe
        ret


;@sp+2= dirección en xmem (long)
```

```
;@sp+6= longitud de los datos (int)

xdumpblk::

        ld a,xpc                            ; salva XPC, debemos agregar 2 a los accesos
        push af                             ; relativos al SP
        ld hl, (SP+6)                       ; Obtiene address (long) en BC:DE
        ld c,l
        ld b,h
        ld hl, (SP+4)
        ex de,hl

        call LongToXaddr                    ; Convierte BC:DE a XMEM + address
        ld xpc,a                            ; DE = address, A = xpc

        ld hl,(sp+8)                        ; hl=longitud del bloque
        ld c,h                              ; bc = "
        ld b,l
        ld a,l
        ex de,hl                            ; HL= address
        or a
        jr nz,xcbf_loop                     ; corrige si b=0 (djnz es 256 iteraciones)
        dec c

xcbf_loop:
        ld a,(hl)                           ; escribe 2 pixels
        ioe ld  (0x8001),a
        inc hl
        ld  a,(hl)
        ioe ld  (0x8001),a
        inc hl
        djnz xcbf_loop                      ; loop en b
        xor a
        dec c
        jp p,xcbf_loop                      ; loop en c

xcbf_done:
        pop af                              ; restablece XPC
        ld xpc,a
        ret

#endasm


#define Read_Data() RdPortE(0x8001)
```

El prefijo *ioe* es el que nos permite utilizar cualquier instrucción de acceso a memoria como instrucción de I/O, en este caso en un espacio de direccionamiento externo (bus). Esta es una de las mayores diferencias entre Rabbit y Z-80, en cuanto a set de instrucciones; descartando, claro está las funciones agregadas.

El resto de las funciones se ha escrito en C, dado que con el incremento de velocidad logrado es suficiente para el módulo utilizado y las prestaciones esperadas de una nota de aplicación. Afortunadamente, el fabricante nos ha provisto un set de rutinas en C que pudimos portar muy fácilmente, las mismas proveen un fácil acceso a los comandos principales y prescindiremos de transcribirlas aquí para no hacer de ésta una nota demasiado extensa.

Veamos ahora la rutina de inicialización:

```
void SSD_init ()
{

// Usamos el Port E bit 4 para el Chip Select con 7 wait-states
#define STROBE         0x10
#define CSREGISTER     IB4CR
#define CSSHADOW       IB4CRShadow
#define CSCONFIG       0x48

        // Inicializo como I/O
        WrPortI(PEFR, &PEFRShadow, (PEFRShadow|STROBE));

        // Inicializo como salida
        WrPortI(PEDDR, &PEDDRShadow, (PEDDRShadow|STROBE));
```

```
        // Inicializo como chip select.
        WrPortI(CSREGISTER, &CSSHADOW, CSCONFIG);

        // Seteo clock a PCLK/2
        WrPortI(PECR, &PECRShadow, (PECRShadow & ~0xFF));
        MsDelay ( 1000 );                   // espero un cierto tiempo


        SetDispOff();
        SetMultiplex(0x3f);                 // 96x64 => MUX =63
        SetStartLine(0x00);                 // display comienza en 0
        SetDispOffset(0x00);
        SetRemap_Format(0x30);              //8-bit color, remapped (para 0,0 arriba)
        SetMasterCurrent(0x0f);             // brillo y contraste standard
        SetContrast_ColorA(0xff);           //40 cd/m^2 @12V=0x15     18cd/m^2 @12V=0x02
        SetContrast_ColorB(0xff);           //
        SetContrast_ColorC(0xff);           //40 cd/m^2 @13V=0x14

        SetVpa_Lv(0x3f);
        SetVpb_Lv(0x3f);
        SetVpc_Lv(0x3f);
        SetVcomh(0x7f);

        SetPowerSave();                     // bajo consumo
        SetPhase12(0x24);
        SetDispClk_Div(0xd0);               // frecuencia de operación
        SetMaster_Config(0x8e);             // generador de Vcc externo
        SetDispOn();
}
```

Como dijéramos anteriormente, utilizamos IOSTROBEs para generar el chip select. La cantidad de wait-states la hemos determinado a partir de la hoja de datos del controlador, tomando en cuenta el reloj del RCM3720, para respetar las condiciones de operación recomendadas en cuanto a tiempo de acceso. Los demás datos los tomamos de la inicialización sugerida por el fabricante.

A continuación, las rutinas de manejo gráfico, despliegue de pantallas y relleno. Para simplificar lo más posible la operación, trabajaremos en 8bpp, es decir, 256 colores. En cambio, a la hora de mostrar imágenes, lo haremos en 16 bpp (65536 colores). La estructura interna de memoria del SSD1332 asigna un formato 3:3:2 para 8bpp y 5:6:5 para 16bpp; a los fines prácticos, dejamos los bits individuales sueltos o los agrupamos, según como nos fuera más conveniente.

```
#define SSD_home() SetColumnAddr(0,95);SetRowAddr(0,63);


void SSD_fill(unsigned char pattern)
{
int i;
        SSD_home();                                 // address 0,0
        i=96*64;
        while(i--)
                Write_Data(pattern);                // fill
}

#define SSD_clear() SSD_fill(0)

#define SSD_plot(x,y,a,b,c) Draw_Line(x,y,x,y,a,b,c)


#ximport "gohan.ssd" gohan
#ximport "dbzgroup.ssd" dbzgroup
#ximport "goku.ssd" goku


void SSD_dump (unsigned long imgdata)
{
unsigned int tocopy,len;

        SSD_home();                                 // address 0,0
        len=96*64;
        imgdata+=sizeof(long);                      // apunta a imagen
        while(len) {                                // manda en bloques de 2K
```

```
                tocopy=2048;
                if(tocopy>len)
                        tocopy=len;
                xdumpblk(imgdata,tocopy);
                imgdata+=(tocopy<<1);
                len-=tocopy;
        }
}
```

Para cargar las imágenes en memoria, utilizamos la directiva *#ximport*, que nos permite tomar un archivo de la computadora, al momento de compilar, e incorporarlo en *xmem* en el módulo Rabbit. Como pudimos observar, para recuperar los datos simplemente direccionamos esa zona, recordando que antes de los datos propiamente dichos hay un '*long*' que contiene la longitud de los mismos, la cual en este caso no utilizamos porque ya la conocemos

Para los sets de caracteres, definimos una simple estructura en memoria que nos permita conocer los parámetros básicos del set, para poder dibujarlo. Para hacer más simple el proceso, limitamos el ancho de los caracteres a 16 pixels como máximo. Aprovechamos los sets de caracteres que ya trae Dynamic C, aunque podríamos generar los nuestros utilizando la utilidad *fbmcnvtr* que viene en el directorio *Utilities* de Dynamic C. Como adelantáramos, escribiremos un byte por cada pixel. Dado que los sets de caracteres están almacenados a 1bpp, deberemos hacer un pequeño procesamiento previo. Como no nos cuesta nada, agregamos la opción de hacer el fondo transparente.

```
#use "6X8L.lib"
#use "12X16L.lib"

typedef struct {
        unsigned long *font;
        unsigned char  lpc;           // líneas por caracter
        unsigned char  Bpcl;          // bytes por línea de caracter (1 ó 2)
        unsigned char  bpcl;          // bits por línea de caracter
} FontInfo;

const static FontInfo fontinfo[]={
{&Font6x8,8,1,6},
{&Font12x16,16,2,12}
};

void SSD_putchr ( int font, char chr , int color, int bcolor)
{
int i,j,ii,jj;
unsigned long address;
int data,aux;

        i=fontinfo[font].lpc;                       // líneas por caracter
        ii=fontinfo[font].Bpcl;                     // bytes por línea
        jj=fontinfo[font].bpcl;                     // bits por línea
        address=*(fontinfo[font].font)+i*ii*(chr-0x20);
        while(i--){
                data=xgetint(address);              // manda todos los datos
                aux=(data&0xFF)<<8;                 // swap bytes
                data=((data>>8)&0xFF)+aux;
                address+=ii;
                j=jj;
                while(j--){
                    if(data&0x8000)                 // caracteres justificados a MSB
                        Write_Data(color);          // 1 => color
                    else                            // 0 => bcolor
                        if(bcolor>=0)               // bcolor =-1 => transparente
                                Write_Data(bcolor);
                        else
                                Read_Data();        // (sólo incrementa address counter)
                    data<<=1;                       // bit siguiente
              }
        }
}

void SSD_printat (int font,unsigned int row,unsigned int col,char *ptr,int color,int bcolor)
{
        do {            // setea área a tamaño de caracter y manda los bytes del caracter
```

```
                SetColumnAddr(col,col+fontinfo[font].bpcl-1);        // bits por línea
                SetRowAddr(row,row+fontinfo[font].lpc-1);           // líneas por caracter
                SSD_putchr (font,*ptr++,color,bcolor);
                col+=fontinfo[font].bpcl;
        } while (*ptr);
}
```

Finalmente, el programa principal, que realiza una pequeña demo para cautivar a los amantes de los aparatitos electrónicos y los dibujos animados.

```
/* MAIN PROGRAM */

main()
{
int i;


        SSD_init();
        Fill(1);                                        // habilita relleno de figuras
        while(1){
                SSD_home();
                SSD_clear();
                Draw_Rectangle(10,10,70,50,7,29,7,20,4,6);   // dibuja un rectángulo lleno
                SSD_printat(1,20,20,"Si!",103,-1);
                MsDelay ( 3000 );                       // espera 3 secs
                SSD_clear();                            // borra pantalla

                for(i=0;i<96;i+=2)
                        SSD_plot(i,32,15,1,1);          // eje x
                for(i=0;i<64;i+=2)
                        SSD_plot(48,i,15,1,1);          // eje y
                for(i=-48;i<48;i++)                  // plot sin(x); 0,0 is at top left
                        SSD_plot(48+i,32-(int)(32*(sin(i*3.14159/48))),2,2,32);
                SSD_printat(0,0,0,"sin(x)",127,10);
                MsDelay ( 3000 );                       // espera 3 secs
                SSD_printat(0,56,0,"Cika Electronica",255,-1);
                MsDelay ( 3000 );                       // espera 3 secs
                SetRemap_Format(0x70);                  // 16-bit color, remapped
                SSD_dump(gohan);
                MsDelay ( 5000 );                       // espera 5 secs
                SSD_dump(goku);
                MsDelay ( 5000 );                       // espera 5 secs
                SSD_dump(dbzgroup);
                MsDelay ( 5000 );                       // espera 5 secs
                SetRemap_Format(0x30);                  // 8-bit color, remapped
        }
}
```