Nota de Aplicación: CAN-091
Título: **Utilización de displays LCD color con controladores SSD1963 y Rabbit**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	11/02/11	port de CAN-035 y CAN-087

Presentamos una forma de utilizar displays TFT de 640x480 en formato VGA, como por ejemplo el WF57FTLBDF0# de Winstar, basados en el controlador SSD1963.

Breve descripción del SSD1963

Hardware

El SSD1963 es un controlador inteligente para displays LCD de alta resolución, se encarga de generar las señales que necesita un display TFT. La configuración de este dispositivo es algo compleja, pero afortunadamente existe información como para resolverlo. La imagen a enviar al display se aloja en una RAM interna de 1,2MB (1215Kbytes), la cual es direccionada por el controlador y no es accesible directamente al usuario. La interfaz entre el SSD1963 y el procesador host puede elegirse entre un formato tradicional como el del legendario Motorola 6800, utilizado mayormente por los displays alfanuméricos y otro tradicional como el del Intel 8080, utilizado por muchos displays gráficos. El bus de datos puede utilizarse de 8- a 24-bits, pero el display empleado presenta una interfaz fija de 8-bits. Si bien el controlador puede funcionar a 3,3V o a tensiones más bajas, el display está especificado para funcionar con procesadores de 3,3 V.

Software

El SSD1963 se encarga de todo lo referente al despliegue de la imagen, la cual reside como dijéramos en su memoria interna (1215KB). Mediante los registros de control, es posible indicar en qué zona de memoria comienza la pantalla y su tamaño. La cantidad de bits asignados a definir el color de cada pixel es de 24; en un bus de 8-bits el procesador escribe tres bytes por cada pixel. Los pixels se distribuyen de arriba a abajo y de izquierda a derecha, conforme avanzan las posiciones de memoria.

Una característica muy interesante de este controlador es que requiere que se le defina un área de trabajo, y automáticamente va llenando dicha área con la información que recibe. Es decir, no es necesario indicarle las direcciones donde van los datos sino que definida el área de trabajo y la dirección de incremento, el llenado del área es automático.

Desarrollo propuesto

Para mantener la simpleza, optamos por utilizar el controlador siempre en el modo por defecto, en el cual las direcciones se incrementan de izquierda a derecha y de arriba a abajo.

Si bien esto se ha realizado sobre Rabbit, es sumamente sencillo portarlo a cualquier otra arquitectura con capacidad suficiente. Una ventaja de utilizar Rabbit 5000 es que al tener bus externo y IOSTROBE permite realizar la escritura del display a velocidad de escritura en memoria (3 ciclos de clock), ya que no se requiere hacer bit-banging para generar las señales del bus Intel como en otos microcontroladores sin capacidad de bus externo. Otra ventaja, comparado con otros microcontroladores de 8-bits, son las instrucciones y registros de 32-bits.

Algoritmos

Para ubicar un punto en pantalla, tenemos una relación directa entre las coordenadas y los valores que pasamos a los registros de configuración, si definimos que la coordenada (0;0) se halla en el extremo superior izquierdo de la pantalla.

Para mostrar pantallas, deberemos agrupar los datos de modo tal de poder enviarlos de forma que llene el área definida en el sentido que el controlador lo espera. Si comparamos la estructura de memoria del display en

su modo por defecto, con la forma de guardar imágenes en 16 colores 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. Además, BMP incluye un encabezado que contiene la paleta de colores.

Por consiguiente, para adaptar una imagen, debemos llevarla a la resolución deseada, espejarla verticalmente, salvarla en formato BMP en 24bpp y por último descartar el encabezado con algún editor hexa. Es conveniente investigar bien el formato BMP y la implementación del software que utilicemos, porque en algunos casos se graba BMP en 32-bits y hemos notado diferencias en los encabezados. No hemos tenido inconvenientes con *Gimp*.

Para desplegar textos, deberemos generar las letras manualmente. La forma más común (y bastante eficiente) de almacenar tipografías en memoria, consiste en agrupar los pixels "pintados" del caracter en bytes, en el sentido horizontal, es decir, un byte aloja ocho pixels que corresponden a la parte superior del caracter, de izquierda a derecha, de MSB a LSB. Si el ancho del caracter es mayor a dieciséis pixels, entonces se utilizarán grupos de dos bytes. Ésta es la forma en la que se alojan los fonts provistos con las libraries de Dynamic C, y con un simple algoritmo los podemos convertir para su uso con un controlador de displays color. Simplemente, chequearemos el estado de cada pixel, y si éste está pintado, lo coloreamos en el display. De igual modo, si no lo está, podemos utilizar un color de fondo.

Hardware de interfaz

Debido a que la especificación del display es de 3V a 3,6V, hemos elegido para esta nota un RCM5600W. Podemos emplear el bus auxiliar de I/O de Rabbit para agilizar la operación.
La conexión al display se realiza como indica el diagrama a continuación:

LCD	***Rabbit***	***función (en Rabbit)***
A0	PE0	pin de I/O
$\overline{RD}$	$\overline{IORD}$	señal de control del bus, $\overline{RD}$ de periféricos
$\overline{WR}$	$\overline{IOWR}$	señal de control del bus, $\overline{WR}$ de periféricos
$\overline{CS}$	PE5	pin de I/o configurado como IOSTROBE, $\overline{CS}$ de acceso a periféricos en direcciones 0xB000 a 0xCFFF
$\overline{RST}$	PE6	pin de I/O
D0	PA0	pin de I/O configurado como bus auxiliar, D0
D1	PA1	pin de I/O configurado como bus auxiliar, D1
D2	PA2	pin de I/O configurado como bus auxiliar, D2
D3	PA3	pin de I/O configurado como bus auxiliar, D3
D4	PA4	pin de I/O configurado como bus auxiliar, D4
D5	PA5	pin de I/O configurado como bus auxiliar, D5
D6	PA6	pin de I/O configurado como bus auxiliar, D6
D7	PA7	pin de I/O configurado como bus auxiliar, D7

Software de bajo nivel

Dado que tendremos bastantes datos para escribir, necesitamos hacerlo rápido, razón por la cual utilizamos assembler para las rutinas críticas. Al llamar a una rutina assembler desde una función C, los parámetros son pasados todos en el stack y el primero además en HL (8 y 16-bits) o en BCDE (32-bits)
La función básica de escritura consiste simplemente en poner un dato en el bus del display. En el Rabbit, escribimos el dato en la posición que utilizará el IOSTROBE y el dato saldrá por el bus auxiliar (puerto A) con un ciclo de escritura:

```
; parámetros
;@sp+2= dato a escribir
```

```
;
LCD_Write::

        ;ld hl,(sp+2)                  ; obtiene dato (LSB), como ya está en HL economizamos
        ld a,l
        ioe ld(0xB000),A
        ret
```

De igual modo, la escritura de un pixel corresponde a escribir tres bytes en el display. Tomamos el dato en un registro de 32-bits, lo partimos en bytes y lo escribimos:

```
; parámetros
;@sp+2= dato a escribir
;
LCD_WritePixel::
        ;ld bcde,(sp+2)        ; obtiene dato (24-bits), como ya está en BCDE economizamos
        ld a,c
        ioe ld(0xB000),A
        ld a,d
        ioe ld(0xB000),A
        ld a,e
        ioe ld(0xB000),A
        ret
```

Los comandos se indican al controlador colocando el pin A0 $(D/\overline{C})$ en estado bajo. Si el comando requiere parámetros, éstos se envían a continuación con dicho pin en estado alto. A tal fin, definimos dos funciones que se entrelazan para resolver la tarea:

```
#define LCD_A0  0
void LCD_WriteCmd(unsigned char cmd)
{
        BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_A0 );  // Baja A0 (Cmd)
        LCD_Write(cmd);
        BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_A0 );  // Sube A0 (Data)
}

void LCD_WriteStrCmd(const unsigned char *cmd,int len)
{
        LCD_WriteCmd(*cmd++);
        while(len--) {
                LCD_Write(*cmd++);
        }
}
```

Retomando el envío de datos, nos queda resolver cómo enviar la información de un área. La tarea consiste en repetidamente enviar la información de cada uno de los pixels, de 24-bits (3 bytes). Para ello escribimos dos rutinas assembler. En un caso, como por ejemplo para borrar el display, necesitamos escribir la información de un color en un área, y en el otro, para mostrar un ícono, copiar información de la memoria al display:

```
; parámetros
;@sp+2= dato a escribir
;@sp+6= cantidad de veces
;
LCD_WriteArea::
        ;ld bcde,(sp+2)        ; obtiene dato (24-bits), como ya está en BCDE economizamos
        ex jkhl,bcde           ; salva en JKHL
        ld bcde,(sp+6)         ; obtiene cuenta (unsigned long, 32-bits)
        ex de,hl               ; parte cuenta en BCDE, 16 LSb en BC y 16 MSb en DE
        ex bc,hl
        ex de,hl
        ld a,c                 ; corrige si BC=0 (DWJNZ se ejecutaría 65536 veces)
        or b
        jr nz,.l2
.l1:
        dec de                 ; loop externo, DE veces
.l2:
        ex jkhl,bcde           ; loop interno, BC veces, recupera dato, salva cuenta
        ld a,c                 ; escribe bytes que lo constituyen
        ioe ld(0xB000),A
        ld a,d
        ioe ld(0xB000),A
```

```
        ld a,e
        ioe ld(0xB000),A
        ex jkhl,bcde            ; vuelve a salvar, recupera cuenta
        dwjnz .l2               ; loop interno, BC veces
        ld a,d
        or e
        jr nz,.l1               ; loop externo, DE veces
        ret


; parámetros
;@sp+2= puntero a área a escribir
;@sp+6= cantidad de pixels
;
LCD_DumpArea::
        ;ld bcde,(sp+2)         ; obtiene puntero, como ya está en BCDE economizamos
        ex jkhl,bcde            ; salva en JKHL
        ld bcde,(sp+6)          ; obtiene cuenta (unsigned long, 32-bits)
        ex de,hl                ; parte cuenta en BCDE, 16 LSb en BC y 16 MSb en DE
        ex bc,hl
        ex de,hl
        ld a,c                  ; corrige si BC=0 (DWJNZ se ejecutaría 65536 veces)
        or b
        jr nz,.l4
.l3:
        dec de                  ; loop externo, DE veces
.l4:
        ex jkhl,bcde            ; loop interno, BC veces, recupera puntero, salva cuenta
        ld px,bcde              ; inicializa puntero de 24-bits en memoria física (lineal)
        ld bcde,(px+0)          ; lee 32-bits (un pixel + 1 byte extra)
        ld a,c                  ; separa bytes
        ioe ld(0xB000),A        ; y envía al display
        ld a,d
        ioe ld(0xB000),A
        ld a,e
        ioe ld(0xB000),A
        ld bcde,px              ; recupera puntero
        ex jkhl,bcde            ; salva en JKHL y obtiene cuenta en BCDE
        ld py,bcde              ; salva cuenta en PY
        ld bcde,3
        add jkhl,bcde           ; incrementa puntero a siguiente pixel, 3 posiciones
        ld bcde,py              ; recupera cuenta
        dwjnz .l4               ; loop interno, BC veces
        ld a,d
        or e
        jr nz,.l3               ; loop externo, DE veces
        ret
```

A continuación, la inicialización del chip. La primera función se encarga de definir el área de trabajo. La segunda realiza la inicialización propiamente dicha:

```
void LCD_setwindow ( unsigned int x0, unsigned int x1, unsigned int y0, unsigned int y1 )
{
        LCD_WriteCmd(0x2A);                             // direcciones de columnas
        LCD_Write(((unsigned)x0>>8)&0xFF);              // HI byte
        LCD_Write(x0&0xFF);                             // LO byte
        LCD_Write(((unsigned)x1>>8)&0xFF);              // HI byte
        LCD_Write(x1&0xFF);                             // LO byte
        LCD_WriteCmd(0x2B);                             // direcciones de filas
        LCD_Write(((unsigned)y0>>8)&0xFF);              // HI byte
        LCD_Write(y0&0xFF);                             // LO byte
        LCD_Write(((unsigned)y1>>8)&0xFF);              // HI byte
        LCD_Write(y1&0xFF);                             // LO byte
}

void LCD_init ()
{
const static unsigned char init_string1[]={
0xE0,0x01                                       // START PLL
};
const static unsigned char init_string2[]={
0xE0,0x03                                       // LOCK PLL
};
```

```
const static unsigned char init_string3[]={
0xB0,                                   // SET LCD MODE  SET TFT 18Bits MODE
0x0C,0x80,                              // SET TFT MODE & hsync+Vsync+DEN MODE
0x02,0x7F,                              // SET horizontal size=640-1
0x01,0xDF,                              // SET vertical size=480-1
0x00                                    // SET even/odd line RGB seq.=RGB
};

const static unsigned char init_string4[]={
0xF0,0x00                               // SET pixel data I/F format=8bit
};

const static unsigned char init_string5[]={
0x3A,0x60                               // SET R G B format = 6 6 6
};

const static unsigned char init_string6[]={
0xE6,0x02,0xFF,0xFF                     // SET PCLK freq=4.94MHz; pixel clock frequency
};

const static unsigned char init_string7[]={
0xB4,                                   // SET HBP,
0x02,0xF8,                              // SET HSYNC Total=760
0x00,0x44,                              // SET HBP 68
0x0F,                                   // SET VBP 16=15+1
0x00,0x00,                              // SET Hsync pulse start position
0x00                                    // SET Hsync pulse subpixel start position
};

const static unsigned char init_string8[]={
0xB6,                                   // SET VBP,
0x01,0xF8,                              // SET Vsync total
0x00,0x13,                              // SET VBP=19
0x07,                                   // SET Vsync pulse 8=7+1
0x00,0x00                               // SET Vsync pulse start position
};


        MsDelay ( 1000 );               // espera
        BitWrPortI ( PEDR, &PEDRShadow, 0,6 );       // baja RST
        MsDelay ( 10 );                              // espera
        BitWrPortI ( PEDR, &PEDRShadow, 1,6 );       // sube RST
        MsDelay ( 100 );                // espera

        LCD_WriteCmd ( 0x01 );          // Software reset
        LCD_WriteCmd ( 0x01 );          // Software reset
        LCD_WriteCmd ( 0x01 );          // Software reset
        MsDelay ( 100 );                // espera
        LCD_WriteStrCmd ( init_string1,sizeof(init_string1) );      // escribe comandos
        LCD_WriteStrCmd ( init_string2,sizeof(init_string2) );
        LCD_WriteStrCmd ( init_string3,sizeof(init_string3) );
        LCD_WriteStrCmd ( init_string4,sizeof(init_string4) );
        LCD_WriteStrCmd ( init_string5,sizeof(init_string5) );
        LCD_WriteStrCmd ( init_string6,sizeof(init_string6) );
        LCD_WriteStrCmd ( init_string7,sizeof(init_string7) );
        LCD_WriteStrCmd ( init_string8,sizeof(init_string8) );
        LCD_setwindow ( 0, 639, 0, 479);     // área de trabajo
        LCD_WriteCmd ( 0x29 );          // Display ON
}
```

Software

El resto del software lo escribimos en C, por comodidad y velocidad de desarrollo. Por ejemplo, para colocar un color en un área del display (o borrarlo):

```
void LCD_fill(unsigned int x, unsigned int y, unsigned long color)
{
        LCD_WriteCmd(0x2C);
        LCD_WriteArea(color,(unsigned long)x * y);   // while(x*y--) LCD_WritePixel(color)
}

#define LCD_clear()    LCD_setwindow(0,639,0,479);LCD_fill(640,480,0)
```

Para mostrar un ícono:

```
void LCD_icon(unsigned int x, unsigned int y, unsigned int wx, unsigned int wy, unsigned long
img)
{
        LCD_setwindow ( x, wx+x-1, y, wy+y-1);
        LCD_WriteCmd(0x2C);
        LCD_DumpArea(img+4,(unsigned long)wx * wy);  // while(wx*wy--) LCD_WritePixel(*img++)
}
```

por ejemplo:

```
        LCD_icon(160,80,320,305,maiaicon);    // muestra maiaicon, de 320x305, en (160;80)
```

Para iluminar un punto de pantalla, definimos un área de 1x1:

```
LCD_plot(unsigned int x,unsigned int y,unsigned long color)
{
        LCD_setwindow (x,x,y,y);
        LCD_WriteCmd(0x2C);
        LCD_WritePixel(color);
}
```

por ejemplo:

```
        LCD_plot(10,20,0x00ff00);     // pone en verde el punto (10;20)
```

El algoritmo de generación de textos podría optimizarse en assembler, sin embargo hemos decidido dejarlo en C para los alcances de esta nota.

Aprovechando que las fonts de Dynamic C están definidas como libraries, las podemos incluir automáticamente en memoria extendida (*xmem*). Definiremos una simple estructura para guardar algunos parámetros de cada tipografía que nos permitan acelerar su impresión, estos datos los obtenemos observando el archivo que contiene la tipografía, que no es otra cosa que código fuente:

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

typedef struct {
        unsigned long *font;
        unsigned char  lpc;           // líneas por character
        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}
};
```

A continuación, veremos una rutina para imprimir un caracter:

```
nodebug void LCD_putchr(unsigned int font,unsigned int row,unsigned int col,char chr
,unsigned long color,long bcolor)
{
int i,j,ii,jj;
unsigned long address;
int data,ddata,aux;

        i=fontinfo[font].lpc;                       // líneas por caracter
        ii=fontinfo[font].Bpcl;                     // bytes por línea de caracter
        jj=fontinfo[font].bpcl;                     // bits por línea de caracter
        address=*(fontinfo[font].font)+i*ii*(chr-0x20);     // ubica bitmap del caracter
        LCD_setwindow (col,col+jj-1,row,row+i-1);   // define área
        LCD_WriteCmd(0x2C);                         // escritura
        while(i--){                                 // loop externo
                data=xgetint(address);                      // obtiene dos bytes, bajo, alto
                aux=(data&0xFF)<<8;                         // swap bytes
                data=((data>>8)&0xFF)+aux;
                address+=ii;
                j=jj;
                while(j--){                                 // los imprime, bit a bit
```

```
                        if(data&0x8000)
                                LCD_WritePixel(color);
                        else {
                                if(bcolor>=0)                   // bcolor <=-1 => no usado
                                        LCD_WritePixel(bcolor);
                                else
                                        LCD_WritePixel(0UL);
                        }
                        data<<=1;
                }
        }
}
```

Para escribir un string en una posición de pantalla, procedemos de la siguiente forma:

```
void LCD_printat(unsigned int font,unsigned int row,unsigned int col,char *ptr,unsigned long
color,long bcolor)
{
        do {
                LCD_putchr (font,row,col,*ptr++,color,bcolor);
                col+=fontinfo[font].bpcl;
        } while (*ptr);
}
```

Para escribir un texto, simplemente llamamos a esta rutina, teniendo cuidado de no excedernos en los límites útiles:

```
        LCD_printat(0,20,20,"Cika Electronica",0x0000ff,0x000000); // azul, fondo negro
```

Finalmente, como punto importante, tengamos en cuenta al inicializar el módulo de setear correctamente los pines bidireccionales en el sentido en que los usamos, y todos en el estado inactivo. Inmediatamente después, inicializamos el chip:

```
#define PORTA_AUX_IO                                            // external I/O bus
// Port E bit 5 como LCD Chip Select con 3 wait-states
#define LCDSTROBE      0x20
#define LCDCSREGISTER  IB5CR
#define LCDCSSHADOW    IB5CRShadow
#define LCDCSCONFIG    0xC8

        WrPortI ( PEDR,&PEDRShadow,'\B01100111' );              // CS,RD,WR -> HIGH
        WrPortI ( PEDDR,&PEDDRShadow,'\B01100111' );            // PE0,1,2,5,6 = output
        WrPortI(PEFR, &PEFRShadow, (PEFRShadow|LCDSTROBE));
        WrPortI(PEDDR, &PEDDRShadow, (PEDDRShadow|LCDSTROBE));
        WrPortI(LCDCSREGISTER, &LCDCSSHADOW, LCDCSCONFIG);
        WrPortI(PECR, &PECRShadow, (PECRShadow & ~0xFF));
```