Cika Electrónica

Nota de Aplicación: CAN-036
Título: **Utilización de displays LCD color con controladores S1D13706 y Rabbit**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	27/06/05	

Modificamos levemente el desarrollo de la CAN-035, para trabajar en modo 8bpp, es decir 256 colores de una paleta de 256K

Algoritmos

Para ubicar un punto en pantalla, calculamos su posición en memoria sabiendo que alojamos un pixel por byte, es decir: $mem = x + 320 \cdot y$.

Para graficar funciones, debemos tener en cuenta 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 aproveche de manera eficiente la estructura de memoria. Si comparamos la estructura de memoria del display con la forma de guardar imágenes en 256 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, reducirla a 256 colores, espejarla verticalmente, salvarla en formato BMP y por último descartar los 1078 bytes del comienzo. Entre los bytes a descartar tomaremos los bytes 54 a 1077, los cuales corresponden a la paleta en formato BGR0 (4 bytes), y la guardaremos como RGB. Dado que esto es algo más tedioso que para 16 colores, se recomienda escribir un pequeño programita que lo haga

Para desplegar textos, valen las mismas consideraciones que en 4bpp, según viéramos en la CAN-035; excepto que el software será más simple

Configuración del S1D13706

Los valores a setear en cada uno de los registros se obtienen de idéntica forma que en la CAN-035, con la salvedad de indicar 8bpp en la solapa *Preferences*. Exportamos de igual modo los datos en un archivo de tipo *C header file* (s1d13706.h), el cual podremos editar e incluir en el código para Rabbit

Software de bajo nivel

Nuevamente tenemos un bus de 16-bits simulado dentro de uno de 8-bits, con señalización similar a la utilizada por el 8086, con la complicación adicional que nuestro bus de direcciones es ahora de 17-bits, y los punteros de nuestro procesador son de 16-bits. Recibiremos entonces la dirección como un *long*, es decir, 32 bits, y emplearemos para A16 un registro adicional (LSB). Obviamente, dado que ahora la cantidad de información es aún mayor, utilizamos assembler para las rutinas críticas.

```
/* VERSIONS FOR LONG ADDRESS */

#asm root
read13706::
      call addressbus               ; pone addresses y BHE, devuelve address I/O en DE
      ex de,hl
      ioe ld a,(hl)                 ; lee port paralelo
      ld h,0
      ld l,a
      ret
```

```
;sp+2: A0-A15
;sp+4: A16
;sp+6: data
write13706::
      call addressbus               ; pone addresses y BHE, devuelve address I/O en DE
      ld hl,(sp+6)                  ; lee dato
      ld a,l
      ex de,hl
      ioe ld (HL),a                 ; pone dato
      ret

;sp+2: dirección de retorno a la función que nos llama
;sp+4: A0-A15
;sp+6: A16
;sp+8: dato
addressbus:
       ld hl,(sp+4)                 ; lee address
       ex de,hl                     ; en DE
       ld hl,(sp+6)                 ; lee address A16
       ld a,l                       ; en A
ab2:   ld hl,PGDR                   ; apunta a port
       ioi ld (hl),d                ; pone parte alta
       ld hl,PDDR                   ; apunta a port
       ioi ld (hl),e                ; pone parte baja (7,6)
       ld hl,PCDR                   ; apunta a port
       ioi res 4,(HL)               ; A16=0
       rrca                         ; A16=1 ?
       jr nc,ok3                    ; no
       ioi set 4,(HL)               ; sí, A16=1
ok3:
       ld a,e                       ; A= A7-A0
       ld hl,PBDR                   ; apunta a port paralelo
       ioi set 0,(hl)               ; BHE=1
       rrc e                        ; debo activar BHE ? (test LSB=1)
       jr nc,ok1                    ; no
       ioi res 0,(HL)               ; sí, BHE=0
ok1:
       and 0x3F                     ; A6,A7 = 0
       ld e,a
       ld d,0x80                    ; I/O = 0x8000 + A5-A0 = 10000000 00xxxxxx
       ret
#endasm
```

Para poder observar algo, deberemos definir los colores, cargando la paleta. La estructura de paleta es idéntica a las utilizadas en 4bpp, excepto que habrá 256 elementos RGB en la paleta. Dado que ahora nuestras direcciones son longs, la escritura de la paleta es levemente diferente:

```
void writepalette(triplet *address)
{
int i;
unsigned char *ptr;

       ptr=&(address->RGB[0]);
       BitWrPortI(PCDR,&PCDRShadow,0,2);                          // M/R=0 => registros
       for(i=0;i<256;i++){
       write13706(0x0AL,*ptr++);
         write13706(0x09L,*ptr++);
         write13706(0x08L,*ptr++);
         write13706(0x0BL,i);
       }
       BitWrPortI(PCDR,&PCDRShadow,1,2);                          // M/R=1 => memoria
}
```

A continuación, la inicialización del chip. Los valores los obtuvimos utilizando el software de configuración provisto por el fabricante, según comentáramos.

```
typedef unsigned short S1D_INDEX;
typedef unsigned char  S1D_VALUE;

typedef struct
{
```

```
    S1D_INDEX Index;
    S1D_VALUE Value;
} S1D_REGS;

const static S1D_REGS aS1DRegs[] =
{
    {0x04,0x00},   // BUSCLK MEMCLK Config Register
    {0x05,0x22},   // PCLK Config  Register
    {0x10,0xD0},   // PANEL Type Register
    {0x11,0x00},   // MOD Rate Register
    {0x12,0x2B},   // Horizontal Total Register
    {0x14,0x27},   // Horizontal Display Period Register
    {0x16,0x00},   // Horizontal Display Period Start Pos Register 0
    {0x17,0x00},   // Horizontal Display Period Start Pos Register 1
    {0x18,0xFA},   // Vertical Total Register 0
    {0x19,0x00},   // Vertical Total Register 1
    {0x1C,0xEF},   // Vertical Display Period Register 0
    {0x1D,0x00},   // Vertical Display Period Register 1
    {0x1E,0x00},   // Vertical Display Period Start Pos Register 0
    {0x1F,0x00},   // Vertical Display Period Start Pos Register 1
    {0x20,0x87},   // Horizontal Sync Pulse Width Register
    {0x22,0x00},   // Horizontal Sync Pulse Start Pos Register 0
    {0x23,0x00},   // Horizontal Sync Pulse Start Pos Register 1
    {0x24,0x80},   // Vertical Sync Pulse Width Register
    {0x26,0x01},   // Vertical Sync Pulse Start Pos Register 0
    {0x27,0x00},   // Vertical Sync Pulse Start Pos Register 1
    {0x70,0x03},   // Display Mode Register
    {0x71,0x00},   // Special Effects Register
    {0x74,0x00},   // Main Window Display Start Address Register 0
    {0x75,0x00},   // Main Window Display Start Address Register 1
    {0x76,0x00},   // Main Window Display Start Address Register 2
    {0x78,0x50},   // Main Window Address Offset Register 0
    {0x79,0x00},   // Main Window Address Offset Register 1
    {0x7C,0x00},   // Sub Window Display Start Address Register 0
    {0x7D,0x00},   // Sub Window Display Start Address Register 1
    {0x7E,0x00},   // Sub Window Display Start Address Register 2
    {0x80,0x50},   // Sub Window Address Offset Register 0
    {0x81,0x00},   // Sub Window Address Offset Register 1
    {0x84,0x00},   // Sub Window X Start Pos Register 0
    {0x85,0x00},   // Sub Window X Start Pos Register 1
    {0x88,0x00},   // Sub Window Y Start Pos Register 0
    {0x89,0x00},   // Sub Window Y Start Pos Register 1
    {0x8C,0x4F},   // Sub Window X End Pos Register 0
    {0x8D,0x00},   // Sub Window X End Pos Register 1
    {0x90,0xEF},   // Sub Window Y End Pos Register 0
    {0x91,0x00},   // Sub Window Y End Pos Register 1
    {0xA0,0x00},   // Power Save Config Register
    {0xA1,0x00},   // CPU Access Control Register
    {0xA2,0x00},   // Software Reset Register
    {0xA3,0x00},   // BIG Endian Support Register
    {0xA4,0x00},   // Scratch Pad Register 0
    {0xA5,0x00},   // Scratch Pad Register 1
    {0xA8,0x00},   // GPIO Config Register 0
    {0xA9,0x80},   // GPIO Config Register 1
    {0xAC,0x00},   // GPIO Status Control Register 0
    {0xAD,0x00},   // GPIO Status Control Register 1
    {0xB0,0x00},   // PWM CV Clock Control Register
    {0xB1,0x00},   // PWM CV Clock Config Register
    {0xB2,0x00},   // CV Clock Burst Length Register
    {0xB3,0x00},   // PWM Clock Duty Cycle Register
};


#define S1DNUMREGS 54
#define S1DMEMSIZE 76

void init13706()
{
int i;

       for(i=1;i<S1DNUMREGS;i++)
              write13706((long)(aS1DRegs[i].Index),aS1DRegs[i].Value);
}
```

Software

El resto del software lo escribimos mayormente en C, por comodidad y velocidad de desarrollo. Se trata de simples y comunes rutinas que no incluiremos aquí para no extender el texto, pero que el lector puede obtener del archivo adjunto con el software, o consultar en cualquiera de las otras notas de aplicación, dado que son muy similares.

Una excepción es el volcado de imágenes, para el cual escribimos una función especial en assembler. Dado el tamaño de una imagen ($320 \times 240 = 76800\, bytes$), necesitamos una función rápida que copie la imagen directamente desde xmem, leyendo por words y escribiendo de a dos bytes en el S1D13706. Básicamente, tomamos la idea de *xmem2root()* y lo adaptamos a nuestros propósitos, por simplicidad usamos una función externa que resuelve el mapeo de dirección absoluta en 20-bits a valores de *XPC* y dirección lógica.

```
#asm root
;IY: A0-A15
;A: A16
;(IX+0): dato pixel izquierdo
;(IX+1): dato pixel derecho
writew13706::
      ld a',a                           ; save A16
      ld hl,iy                          ; A15-A0
      ex de,hl                          ; en DE
      call ab2                          ; addresses y BHE
      ld a,(IX+0)                       ; lee dato
      ex de,hl
      ioe ld (HL),a                     ; pone dato
      ex af,af'                         ; restore A16
      ld hl,iy                          ; lee address
      ld de,0x0001                      ; next byte
      add hl,de                         ; inc address
      adc a,d                           ; (adc a,0x00) propaga carry
      ex de,hl                          ; A15-A0 in DE
      ld a',a                           ; save A16
      call ab2                          ; address bus
      ex de,hl
      ld a,(IX+1)                       ; lee dato
      ioe ld (HL),a                     ; pone dato
      ex af,af'                         ; restore A16
      ret

xdumpblk::
       ld a,xpc                         ; salva frame pointer, y xpc
       push   af
       push   ix

       ld hl,(sp+12)                    ; xmem address (long) en BC:DE
       ld c,l
       ld b,h
       ld hl,(sp+10)
       ex de,hl

              ; Convierte BC:DE a XPC y address
       call LongToXaddr                 ; DE = logical address, A = xpc
       ld xpc,a
       ex de,hl
       ld ix,hl                         ; IX = source


       ld iy,(sp+6)                     ; a:iy=dest, hl=length
       ld hl,(sp+8)
       ld a,l
       ld hl,(sp+14)

       ld c,h                           ; c:b=length
       ld b,l
       ex af,af'
       ld a,l
       or a
       jr nz,.xcbf_2
       dec c
```

```
.xcbf_2:
        ex af,af'

xcbf_loop:
        call writew13706                    ; preserva BC,IX,IY,A
        ld de,0x0002
        add ix,de
        add iy,de
        adc a,d                             ;(adc a,0x00) propaga carry
        djnz xcbf_loop
        dec c
        jp p,xcbf_loop

xcbf_done:
        pop ix
        pop af
        ld xpc,a
        ret
#endasm


root useix void LCD_dump(long imgdata, triplet *paldata)
{

long dest;
unsigned int tocopy,len;


        len=S1DMEMSIZE/2;
        dest=0;
        imgdata+=sizeof(long);
        writepalette(paldata);
        while(len) {
                tocopy=2048-(int)(dest&2047);
                if(tocopy>len)
                        tocopy=len;
                xdumpblk(dest,imgdata,tocopy);
                dest+=(tocopy<<1);
                imgdata+=(tocopy<<1);
                len-=tocopy;
        }
}
```

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, o dejarlo sin modificar.

Aprovechando que las fonts de Dynamic C están definidas como libraries, las podemos incluir automáticamente en 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 en 8bpp:

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

	i=fontinfo[font].lpc;                                   // líneas por character
	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 caracter en tipografía
	dispaddress=daddress;                                   // apunta a display
	while(i--){
		data=xgetint(address);                          // obtiene 2 bytes de xmem
		aux=(data&0xFF)<<8;                             // swap bytes (MSB - LSB)
		data=((data>>8)&0xFF)+aux;
		address+=ii;
		j=jj;
		while(j--){                                     // rota a izquierda y chequea
			if(data&0x8000)
				write13705(dispaddress,color);  // pinta
			else
				if(bcolor>=0)           // bcolor =-1 => transparente
					write13705(dispaddress,bcolor);
			data<<=1;
			dispaddress++;
		}
		dispaddress=daddress+=320;                      // apunta a línea de abajo
	}
}
```

El cálculo de la dirección inicial en pantalla para cada caracter de un string, la realizamos de la siguiente forma:

```
void LCD_printat (int font,unsigned int row,unsigned int col,char *ptr,int color,int bcolor)
{
unsigned long address;

	address=320L*row;                                       // ubica dirección en display
	do {
		address+=col;
		LCD_putchr (font,address,*ptr++,color,bcolor);  // pone caracter
		col=fontinfo[font].bpcl;                        // siguiente
	}	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",255,6);
```