Nota de Aplicación: CAN-094
Título: **Utilización de displays LCD color con controladores SSD1963 y Holtek ARM Cortex-M3**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	14/10/11	port de CAN-091

Presentamos una forma de utilizar displays TFT de 640x480 en formato VGA, como por ejemplo el WF57FTLBDF0# de Winstar, basados en el controlador SSD1963. En esta oportunidad los conectamos al HT32F1253 de Holtek, un micro con core ARM Cortex-M3.

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.

Breve descripción del HT32F1253

La familia HT32F125x de Holtek es una serie de micros de 32-bits con core ARM Cortex-M3, orientada a propósitos generales. El HT32F1253 posee 32KB flash y 8K RAM, junto con una serie de periféricos interesantes que comparten los 48 pines del encapsulado LQFP con los 2 ports de I/O de 16-bits. Puede operar hasta a 72MHz.

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.

La utilización de un procesador de 32-bits, en este caso un ARM Cortex-M3, nos da cierta comodidad en el manejo de grandes cantidades de información como son las imágenes en color.

El empleo del standard CMSIS nos permite poder cambiar de compilador sin mayores modificaciones al código.

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. 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

La especificación del display es de 3V a 3,6V. La conexión al mismo se realiza como indica el diagrama a continuación:

LCD	***MB9BF500R***
A0	PB12
$\overline{RD}$	PB13
$\overline{WR}$	PB14
$\overline{CS}$	PB15
$\overline{RST}$	PB11
D0	PA0
D1	PA1
D2	PA2
D3	PA3
D4	PA4
D5	PA5
D6	PA6
D7	PA7

Software de bajo nivel

Dado que tendremos bastantes datos para escribir, necesitamos hacerlo rápido. Venciendo la tentación de utilizar assembler, escribiremos de modo particular las rutinas críticas, indicándole al compilador que debe optimizar.

La función básica de escritura consiste simplemente en poner un dato en el bus del display. Escribimos el dato en el registro de I/O y luego simulamos la operatoria del bus del 8080 haciendo bit-banging. Gracias a que los

ports están en el area de bit banding, esto resultará en instrucciones con acceso de bits, más rápidas. La implementación de CMSIS de Holtek, al momento de escribir esta nota, no incluye macros para poder acceder a los pines de I/O bit a bit. Sin embargo, esto es muy simple de resolver mediante algunas macros que aplican la definición standard del área de bit banding y el álgebra correspondiente.

```
/* LCD control signals */
#define LCD_RSTb 11
#define LCD_RDb        13
#define LCD_A0b        12
#define LCD_WRb        14
#define LCD_CSb        15

// Bit-band alias = Bit-band base + (byte offset * 32) + (bit number * 4)
/* Bit-Band for Device Specific Peripheral Registers */
#define BITBAND_PERI(addr, bitnum) (HT_PERIPH_BB_BASE + (((uint32_t)(addr) - HT_PERIPH_BASE)
<< 5) + ((uint32_t)(bitnum) << 2))

#define LCD_RST (*((__IO uint32_t *)BITBAND_PERI(&HT_GPIOB->DOUTR,LCD_RSTb)))
#define LCD_RD  (*((__IO uint32_t *)BITBAND_PERI(&HT_GPIOB->DOUTR,LCD_RDb)))
#define LCD_A0  (*((__IO uint32_t *)BITBAND_PERI(&HT_GPIOB->DOUTR,LCD_A0b)))
#define LCD_WR  (*((__IO uint32_t *)BITBAND_PERI(&HT_GPIOB->DOUTR,LCD_WRb)))
#define LCD_CS  (*((__IO uint32_t *)BITBAND_PERI(&HT_GPIOB->DOUTR,LCD_CSb)))

void __INLINE LCD_Write(uint8_t byte)
{
        HT_GPIOA->DOUTR = byte;              // data out
        LCD_CS=0;
        LCD_WR=0;
        LCD_WR=1;
        LCD_CS=1;
}
```

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. Dado que estamos haciendo bit-banging, cada movimiento de una señal de control del display requiere varios preciosos ciclos del procesador. A fin de acelerar el proceso, dado que una imagen en color implica muchas repeticiones de esta función, vamos a dejar el pin $\overline{CS}$ en estado bajo y realizar una escritura rápida:

```
void __INLINE LCD_WriteFast(uint8_t byte)
{
        HT_GPIOA->DOUTR = byte;              // data out
        LCD_WR=0;
        LCD_WR=1;
}

void __INLINE LCD_WritePixel(uint32_t rgb)
{
        LCD_CS=0;
        LCD_WriteFast(((uint32_t)rgb>>16)&0xFF);
        LCD_WriteFast(((uint32_t)rgb>>8)&0xFF);
        LCD_WriteFast(((uint32_t)rgb>>0)&0xFF);
        LCD_CS=1;
}
```

Debido al __INLINE, el compilador incluirá el código en vez de insertar una llamada a subrutina. Obsérvese como accedemos al port de I/O como una estructura. CMSIS define una estructura por cada port de I/O, en la cual cada registro de control o datos del port es un elemento. Aquí, `HT_GPIOA->DOUTR` apunta al registro de datos de escritura.

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:

```
void LCD_WriteCmd(uint8_t cmd)
{
        LCD_A0=0;                    // Lower A0 (Cmd)
        LCD_Write(cmd);
```

```
        LCD_A0=1;                       // Rise A0 (Data)
}

void LCD_WriteStrCmd(const uint8_t *cmd,unsigned 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. 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:

```
void LCD_WriteArea(uint32_t rgb,uint32_t area)
{
        LCD_CS=0;
        while(area--){
                LCD_WriteFast(((uint32_t)rgb>>16)&0xFF);     // LCD_WritePixel(rgb)
                LCD_WriteFast(((uint32_t)rgb>>8)&0xFF);
                LCD_WriteFast(((uint32_t)rgb>>0)&0xFF);
        }
        LCD_CS=1;
}

void LCD_DumpArea(uint8_t *img,uint32_t area)
{
        LCD_CS=0;
        while(area--){
                LCD_WriteFast(*img++);
                LCD_WriteFast(*img++);
                LCD_WriteFast(*img++);
        }
        LCD_CS=1;
}
```

A continuación, la inicialización del chip. La primera función se encarga de definir el área de trabajo. La segunda realiza la inicialización propiamente dicha. Veremos la forma de realizar el delay más adelante.

```
void LCD_setwindow ( uint16_t x0, uint16_t x1, uint16_t y0, uint16_t y1 )
{
        LCD_WriteCmd(0x2A);                                  // SET column address
        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);                                  // SET page address
        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 uint8_t init_string1[]={
0xE0,0x01             // START PLL
};
const static uint8_t init_string2[]={
0xE0,0x03             // LOCK PLL
};

const static uint8_t 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 uint8_t init_string4[]={
0xF0,0x00              //SET pixel data I/F format=8bit
};

const static uint8_t init_string5[]={
0x3A,0x60              // SET R G B format = 6 6 6
};

const static uint8_t init_string6[]={
0xE6,0x02,0xFF,0xFF    //SET PCLK freq=4.94MHz  ; pixel clock frequency
};

const static uint8_t 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 uint8_t 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
};


        Delay ( 1000 );
        LCD_RST=0;                                  // Baja RST
        Delay ( 10 );
        LCD_RST=1;                                  // Sube RST
        Delay ( 100 );

        LCD_WriteCmd ( 0x01 );                      // Software reset
        LCD_WriteCmd ( 0x01 );                      // Software reset
        LCD_WriteCmd ( 0x01 );                      // Software reset
        Delay ( 100 );
        LCD_WriteStrCmd ( init_string1,sizeof(init_string1) );
        LCD_WriteStrCmd ( init_string2,sizeof(init_string2) );
        Delay ( 10 );
        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);
        LCD_WriteCmd ( 0x29 );                      // Display ON
}
```

Software

El resto del software también lo escribimos en C. Por ejemplo, para colocar un color en un área del display (o borrarlo):

```
void LCD_fill(uint16_t x, uint16_t y, uint32_t fcolor)
{
        LCD_WriteCmd(0x2C);
        LCD_WriteArea(fcolor,(uint32_t)x * y);
}

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

Para mostrar un ícono:

```
void LCD_icon(uint16_t x, uint16_t y, uint16_t wx, uint16_t wy, uint8_t *img)
{
        LCD_setwindow ( x, wx+x-1, y, wy+y-1);
        LCD_WriteCmd(0x2C);
        LCD_DumpArea(img,(uint32_t)wx * wy);
```

```
}
```

por ejemplo:

```
        LCD_icon(160,80,320,305,(uint8_t *)maiaicon);        // muestra maiaicon, de 320x305,
                                                             // en (160;80)
```

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

```
void LCD_plot(uint16_t x,uint16_t y,uint32_t pcolor)
{
        LCD_setwindow (x,x,y,y);
        LCD_WriteCmd(0x2C);
        LCD_WritePixel(pcolor);
}
```

por ejemplo:

```
//                R G B
#define RED    0xff0000
#define GREEN  0x00ff00
#define BLUE   0x0000ff
#define BLACK  0x000000
#define WHITE  0xffffff
#define GRAY   0x808080
#define CYAN   0xffff00

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

El algoritmo de generación de textos utiliza fonts de 8 ó 16-bits. 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:

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

typedef struct {
        uint8_t *font;
        uint8_t        lpc;           // lines per character
        uint8_t        Bpcl;          // bytes per character line (1 or 2)
        uint8_t        bpcl;          // bits per character line
} FontInfo;

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

A continuación, veremos una rutina para imprimir un caracter:

```
void LCD_putchr(uint16_t font,uint16_t row,uint16_t col,char chr ,uint32_t fcolor,long
bcolor)
{
int i,j,ii,jj;
uint8_t *address;
uint16_t data;

        i=fontinfo[font].lpc;                          // lines per character
        ii=fontinfo[font].Bpcl;                        // bytes per character line
        jj=fontinfo[font].bpcl;                        // bits per character line
        address=&(fontinfo[font].font[i*ii*(chr-0x20)]);
        LCD_setwindow (col,col+jj-1,row,row+i-1);
        LCD_WriteCmd(0x2C);
        while(i--){
                data=*address;                         // read first byte
                data<<=8;                              // move to high part
                data+=address[1];             // read (possible) second byte in low part
                address+=ii;
                j=jj;
                while(j--){
                        if(data&0x8000)
                                LCD_WritePixel(fcolor);
```

```
                        else {
                                if(bcolor>=0){          // bcolor =-1 => transparent
                                        LCD_WritePixel((uint32_t)bcolor);
                                }
                                else {
                                        LCD_WritePixel(BLACK);
                                }
                        }
                        data<<=1;
                }
        }
}
```

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

```
void LCD_printat(uint16_t font,uint16_t row,uint16_t col,char *ptr,uint32_t fcolor,long
bcolor)
{
        do {
                LCD_putchr (font,row,col,*ptr++,fcolor,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",BLUE,BLACK); // azul, fondo negro
```

En la función de inicialización vimos que utilizábamos una llamada a función para lograr una demora. La misma la realizamos en base al SysTick Timer. Generamos una función cuyo nombre corresponde al standard definido en CMSIS, y el compilador la identificará para utilizarla. Dicha función incrementa una variable, que luego podemos chequear. En este caso lo hacemos en un loop porque no tenemos otra cosa que hacer y nos simplifica el programa en general.

```
volatile unsigned long SysTickCnt;      // SysTick Counter

// SysTick Setup
void SysTick_Setup (void)
{
  SysTick->LOAD = 72000 - 1;            // 1ms: 72000 ticks @ 72MHz
  SysTick->CTRL = 0x0007;               // Habilita Timer y Tick Interrupt
}

// SysTick Interrupt Handler (tick cada 1ms)
void SysTick_Handler (void)
{
  SysTickCnt++;
}

void Delay (unsigned long tick)
{
unsigned long systickcnt;

  systickcnt = SysTickCnt;
  while ((SysTickCnt - systickcnt) < tick);
}
```

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:

```
        HT_CKCU->APBCCR0 |= (1<<17)+(1<<16);         // PBEN, PAEN: APB clocks GPIOA y GPIOB
        HT_GPIOA->DIRCR = 0x00FF;                    // PA.0-7 Outputs
        HT_GPIOB->DIRCR = 0xF800;                    // PB.11,12,13,14,15 Outputs
        HT_GPIOB->DOUTR |= (1<<LCD_CSb)+(1<<LCD_WRb)+(1<<LCD_RDb)+(1<<LCD_RSTb);

        SysTick_Setup();                             // SysTick Setup
```

Los nombres de los pines de I/O corresponden a CMSIS v2.0. Nótese que `HT_GPIOA->DIRCR` apunta al registro de control del port A, en el cual configuramos los pines como salida o como entrada. Por defecto, los clocks hacia cada periférico se encuentran inhabilitados, por lo que antes de utilizar un puerto de I/O debemos habilitarle el clock correspondiente en el bus de periféricos (APB), lo cual realizamos con el registro `HT_CKCU->APBCCR0`.