Build a Smarter World

EC2x-QuecOpen Kernel

Adding SPI Multicolored Display Guide

LTE Standard Module Series

Rev. EC2x-QuecOpen_Kernel_Adding_SPI_Multicolored_Display_Guide_V1.2

Date: 2019-03-14

Status: Preliminary

Our aim is to provide customers with timely and comprehensive service. For any assistance, please contact our company headquarters:

Quectel Wireless Solutions Co., Ltd.
7th Floor, Hongye Building, No.1801 Hongmei Road, Xuhui District, Shanghai 200233, China
Tel: +86 21 5108 6236
Email: info@quectel.com

Or our local office. For more information, please visit:
http://www.quectel.com/support/sales.htm

For technical support, or to report documentation errors, please visit:
http://www.quectel.com/support/technical.htm
Or email to: support@quectel.com

GENERAL NOTES

QUECTEL OFFERS THE INFORMATION AS A SERVICE TO ITS CUSTOMERS. THE INFORMATION PROVIDED IS BASED UPON CUSTOMERS' REQUIREMENTS. QUECTEL MAKES EVERY EFFORT TO ENSURE THE QUALITY OF THE INFORMATION IT MAKES AVAILABLE. QUECTEL DOES NOT MAKE ANY WARRANTY AS TO THE INFORMATION CONTAINED HEREIN, AND DOES NOT ACCEPT ANY LIABILITY FOR ANY INJURY, LOSS OR DAMAGE OF ANY KIND INCURRED BY USE OF OR RELIANCE UPON THE INFORMATION. ALL INFORMATION SUPPLIED HEREIN IS SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.

COPYRIGHT

About the Document

History

Revision	Date	Author	Description
1.0	2018-05-16	Matthew MA	Initial
1.1	2018-05-16	Matthew MA	Added user guide
1.2	2019-03-14	Matthew MA	Updated document's format and contents

Contents

Figure Index

1 Introduction

This document mainly introduces how to add and adapt a SPI LCD under Open Linux and how to configure it according to the selected model from user's develop perspective. This document will help users to do interface programming so that users can develop easy and quickly.

This document mainly applies to global market, and supports the LTE Standard modules currently included:

- EC2x: EC20 R2.1/EC25/EC21

2 Introduction of SPI LCD Driver

SPI is the abbreviation of Serial Peripheral Interface. It is a kind of communication bus with high speed, full duplex and synchronization. It only occupies 4 pins on chip, which saves pin source and also saves space and provides convenience to PCB layout. More and more chips have integrated this communication protocol due to its simple and practical features, such as AT91RM9200.

TFT LCD provides semiconductor switch for each pixel, and its processing is similar to large scale integrated circuit. As each pixel can be controlled directly by electric pulse, so each node is independent comparatively and can be controlled consecutively. Thus, not only accelerated the react speed of the display, but also provided precise control of gray level displaying, so TFT LCD has more vivid colors.

SPI display combined features of SPI and TFT, which enables CPU to control the display with minimum 5 lines. It features fewer port occupation and medium speed of display refreshing. SPI display can present simple characters, Chinese characters, pictures and etc. flexibly, and applies to various display scenarios with low demand of speed.

2.1. Introduction of SPI Driver under Linux

Software architecture of SPI driver in Kernel has done reasonable layering and abstraction as following:

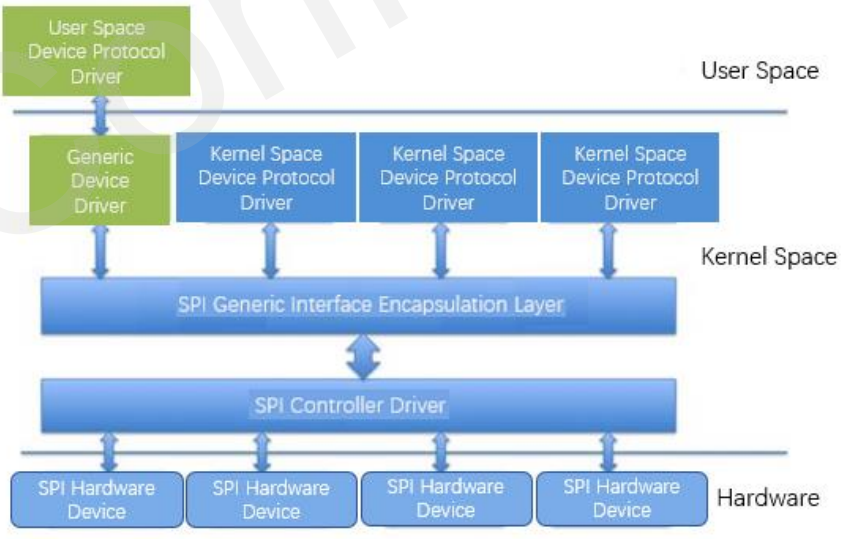


Figure 1: Software Architecture of SPI Driver in Kernel

- SPI controller driver

SPI controller does not need to concern about the specific functions of the device, it is only responsible to send data that prepared by upper layer protocol driver to SPI device according to timing requirement of SPI bus, and report data collected from slave device to upper layer protocol driver at the same time. So, Kernel makes SPI controller driver independent. SPI controller driver controls specific controller hardware, such as DMA and interruption, etc. Because multiple upper layer protocol driver might request data transmission via controller, that's why SPI controller driver also has to manage the requesting queue and implement the principle of first in first out.

In order to simplify the programing of SPI driver and lower the coupling extent of protocol driver and controller driver, Kernel encapsulate some generic operations of controller driver and protocol driver into a standard interface, plus some other generic logistic operations, all above composed SPI generic interface encapsulation layer. Here are advantages: controller driver only needs to implement standard interface callback API and register it to generic interface layer without communicating with protocol driver directly; protocol driver can complete registration of device and driver and data transmission via API provided by generic interface layer without concerning about implementation details of SPI controller driver.

- SPI protocol driver

As mentioned above, controller driver doesn't clear and care about the specific functions of device, SPI protocol driver completes the SPI device's functions, SPI protocol driver knows device's function and communication data's protocol format. Downward, protocol driver exchanges data with controller via generic interface layer; upward, protocol driver usually communicates with Kernel's other subsystems based on device's specific function. For example, communicating with MTD layer to make storage device of SPI a certain file system; communicating with TTY subsystem to make SPI device a TTY device; communicating with network subsystem to make one SPI device a network device. If it is a dedicated SPI device, dedicated protocol driver can be realized according to device protocol.

- SPI generic device driver

Sometimes, due to the variability of device that connected to SPI controller and the corresponding protocol driver hasn't been configured in Kernel, Kernel has prepared generic SPI device driver which provides user space a control interface to control SPI controller, and specific protocol control and data transmission will be done by user space according to specific device. In this way, communication with SPI device can only be completed by synchronization method, so it usually be used for simple SPI device with less data.

As we know, Linux SPI can be divided into 4 parts, in which, SPI controller driver and SPI generic device driver are provided by CPU chip supplier, SPI generic interface encapsulation layer is provided by Linux Kernel source codes, SPI protocol driver is provided by specific device maker (programming by oneself is needed if don't have), the SPI TFT driver this document introduced is SPI protocol driver.

Downloaded SPI controller driver and SPI generic interface encapsulation layer (this layer must exists after SPI controller driver downloading) are needed by SPI protocol driver, then SPI protocol driver downloading process is mainly to mount device and corresponding driver to SPI bus, driver part mainly do serialization of device, the function that sends and receives SPI data is provided by SPI controller driver

and SPI generic interface encapsulation layer.

2.2. Introduction of Framebuffer Driver under Linux

Framebuffer (hereinafter referred to as FB) driver is LCD driver framework under Linux, and also a virtual device in Linux Kernel, it provides a display device with unified standard interface to application layer. FB is an interface provided by Linux to display device and also a device after memory abstraction, it allows upper layer programs to read and write display buffer directly under picture mode. This operation is abstract and unified. Users needn't to care about details such as the physical memory address and paging mechanism, all of that were completed by framebuffer device driver.

Framebuffer usually located in */dev/fb0* (Android platform located in */dev/graphics/fb0*, and they are the same), it provides unified GPU description to upper layer. The first thing needs to be clear is that LCD subsystem is complexed, but it's still a basic character's device, *fb0* is the device node and master device number is 29. The difference is that address mapping can be done through MMAP (MMAP mapping one file or other objectives to internal storage), and mapping video memory space in Kernel to user space, thus the data entered in user space that needs to display can show on LCD directly. Other parameters searching and setting can be completed through *ioctl.*

3 Adding Hardware SPI Display

SPI display usually includes one TFT display, one control chip (ILI9341 is used in this document) and one AD chip (it is used for touch but not necessary), CPU controls the display with SPI bus and 2 GPIO lines. Usually the LCD operation of SPI device driver and GPIO pins selection need to be modified if users have selected different controller chips and GPIOs.

3.1. Hardware connection

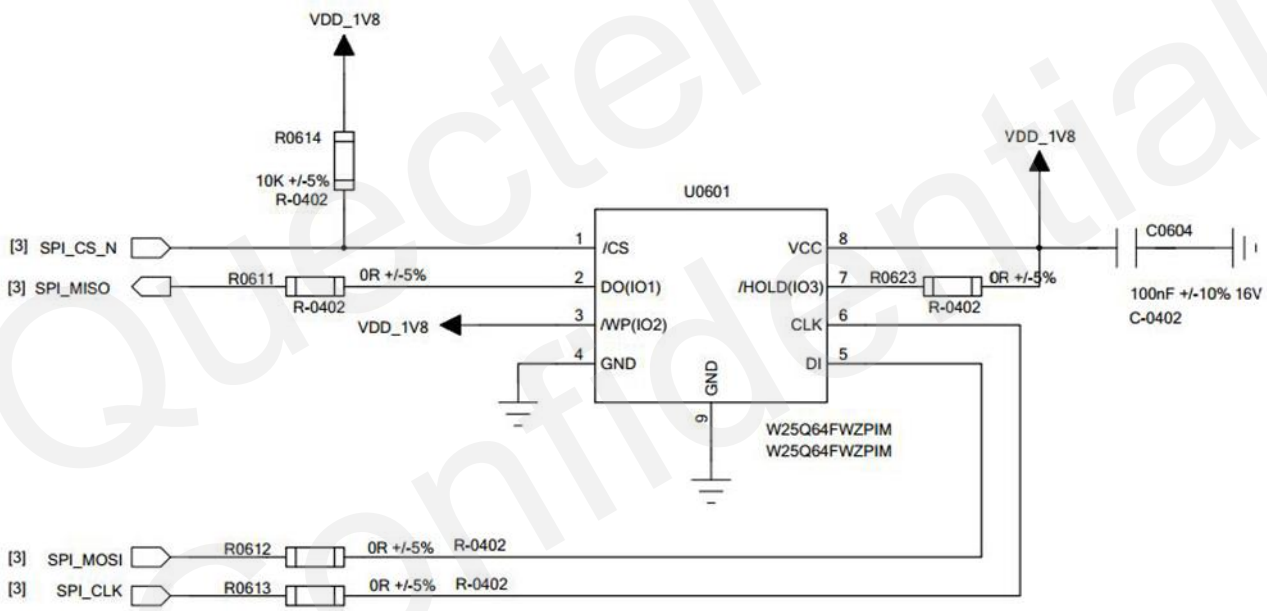


Figure 2: Reference Design of Standard 4 Lines SPI Connecting to TFT

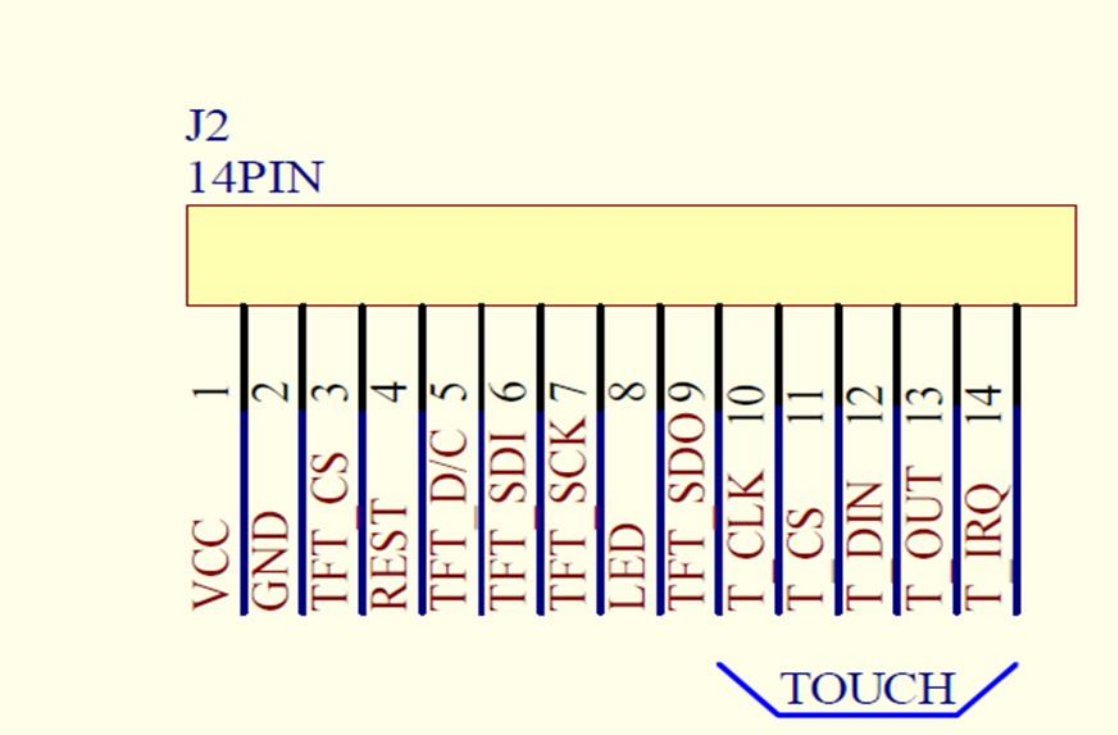


Figure 3: Reference Design of TFT Display Pins

SPI display usually needs 3 power lines and 5 communication lines, in which, 3 power lines included one ground line GND, one controller power line VCC and one backlight line LED; the communication lines included one REST line (GPIO line that can reset SPI display controller and connect to CPU, such as GPIO59), one RS line (it also known as D/C line, a GPIO line that can distinguish between data and command in transmission, and connect to CPU, such as GPIO38), and 3 SPI lines (display CLK line is connected to CPU SPI's CLK, SDI to CPU SPI's MOSI line and CS line to CPU SPI's CS.)

NOTE

There is no need to connect SDO on display as CPU usually do not read from SPI display.

4 Adding SPI LCD Driver under Linux

4.1. SPI Bus Driver Configuration under Linux

SPI LCD driver may use SPI main controller driver, so customers need to configure SPI bus to Linux. In the examples in this document, SPI LCD is mounted on SPI6 bus. There is two parts to configure SPI bus driver.

- SPI configuration of DTS

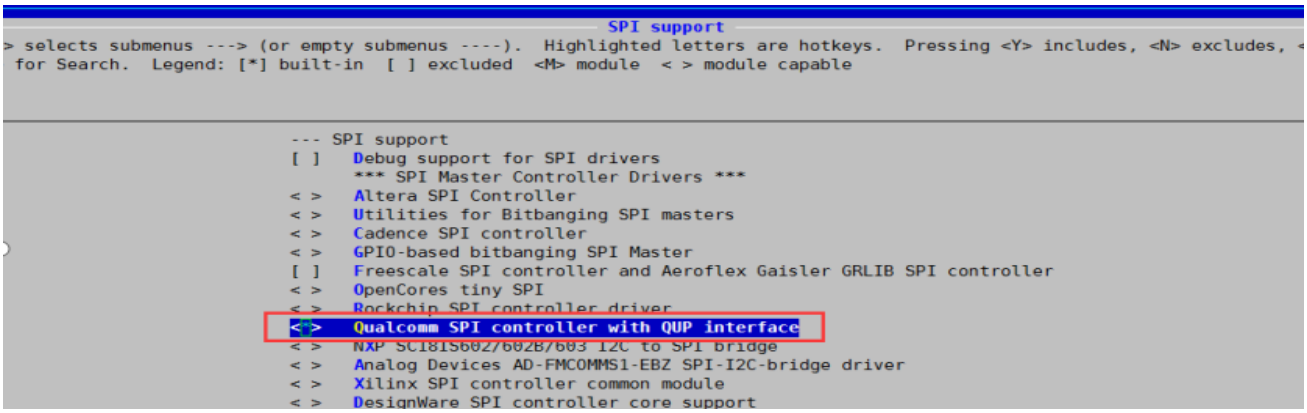
Please find the *mdm9607-mtp.dtsi* of *arch/arm/boot/dts/qcom/* under Kernel directory, this file is the external master switch. Then locate the SPI node with serial number 6, the SPI will be opened if set the "status" as "OK"; and the UART node with serial number 6, set the "status" as disabled. Now, GPIO20-23 will work as SPI. Please check following:

```
00049:
00050: &spi_6 {
00051:     status = "ok";
00052: };
00053:
00054: &blsp1_uart3 {
00055:     status = "ok";
00056: };
00057:
00058: &blsp1_uart2 {
00059:     status = "ok";   //if need, user can enable by themselves
00060:     pinctrl-names ="sleep", "default";
00061:     pinctrl-0 = <&blsp1_uart2_sleep>;
00062:     pinctrl-1 = <&blsp1_uart2_active>;
00063: };
00064:
00065: &blsp1_uart6 {
00066:     status = "disabled";
00067: };
```

- SPI configuration of driver

Please execute the following under SDK directory:

```
$ make kernel_menuconfig
```



Please ensure the configuration as the red box in the above picture. Configure the SPI bus driver to make EC2x platform can drive SPI bus.

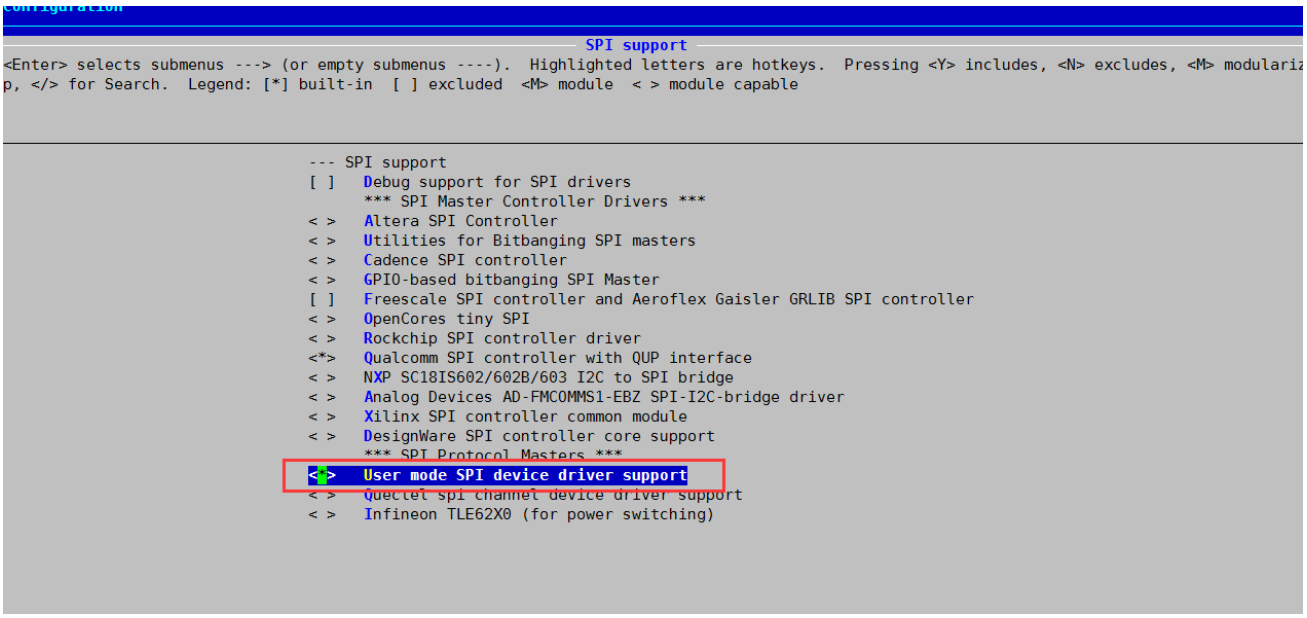
4.2. Adding LCD Driver under Linux

Now, users have SPI controller driver and SPI core driver under Linux. Now, SPI TFT device and SPI TFT device driver are need to be added, the codes of these two parts can be downloaded from

spidev.c

, please copy it to the directory *driver/spi/* under Linux Kernel root directory. Then execute the following under SDK:

```
# make kernel_menuconfig
```



NOTE

There is *spidev.c* under *Driver/spi* directory, but it needs to be replaced with *spidev.c* that supports LCD. Please generate a new *spidev.c* that supports LCD by compiling *spidev.c* with original *Kconfig* and *Makefile* of Kernel.

5 Analysis of SPI LCD Driver

Linux SPI can be divided into 4 parts, in which, SPI controller driver and SPI generic device driver are provided by CPU chip supplier, SPI generic interface encapsulation layer is provided by Linux Kernel source codes, SPI protocol driver is provided by specific device maker (programming by oneself is needed if don't have), the SPI TFT driver this document introduced is SPI protocol driver.

For SPI protocol driver, it requires the system already loaded SPI controller driver and SPI common interface encapsulation layer (this layer must be existed after SPI controller driver loading), then the loading process of SPI protocol driver is mainly to mount device and corresponding driver to SPI bus, driver mainly do serialization of device, the function that sends and receives SPI data is provided by SPI controller driver and SPI generic interface encapsulation layer.

5.1. Analysis of SPI TFT Display Driver (No Need of Transplantation)

SPI TFT display belongs to SPI protocol driver, and here is the driver's main procedure:

- Provide character driven operational function set and create class file

Providing character driven operational function set is mainly to perform read and write to SPI display and provide the operation function.

```
00978:
00979: static int __init spidev_lcd_init(void)
00980: {
00981:     int status;
00982:     /* Claim our 256 reserved device numbers.  Then register a class
00983:      * that will key udev/mdev to add/remove /dev nodes.  Last, register
00984:      * the driver which manages those device numbers.
00985:      */
00986:     BUILD_BUG_ON(N_SPI_MINORS > 256);
00987:     status = register_chrdev(SPIDEV_MAJOR, "spi_lcd", &spidev_lcd_fops);
00988:     if (status < 0)
00989:         return status;
00990:
00991:     spidev_lcd_class = class_create(THIS_MODULE, "spidev_lcd");
00992:     if (IS_ERR(spidev_lcd_class)) {
00993:         status = PTR_ERR(spidev_lcd_class);
00994:         goto error_class;
00995:     }
00996:
```

- Register SPI TFT device and SPI TFT driver on SPI bus.

```
00997:      status = spi_register_driver(&spidev_lcd_spi_driver);
00998:      if (status < 0)
00999:          goto ↓error_register;
01000:
01001:      if (busnum != -1 && chipselect != -1) {
01002:          struct spi_board_info chip = {
01003:                      .modalias    = "spidev_lcd",
01004:                      .mode        = spimode,
01005:                      .bus_num     = busnum,
01006:                      .chip_select     = chipselect,
01007:                      .max_speed_hz    = maxspeed,
01008:          };
01009:
01010:          struct spi_master *master;
01011:
01012:          master = spi_busnum_to_master(busnum);
01013:          if (!master) {
01014:              status = -ENODEV;
01015:              goto ↓error_busnum;
01016:          }
01017:          spi = spi_new_device(master, &chip);
01018:          if (!spi) {
01019:              status = -EBUSY;
01020:              goto ↓error_mem;
01021:          }
```

- Initialize TFT display

SPI TFT device matches SPI TFT driver on SPI bus, and then execute driver's probe function:

```
00960:
00961: static struct spi_driver spidev_lcd_spi_driver = {
00962:      .driver = {
00963:          .name =     "spidev_lcd",
00964:          .owner =    THIS_MODULE,
00965:          .of_match_table = of_match_ptr(spidev_lcd_dt_ids),
00966:      },
00967:      .probe =    spidev_lcd_probe,
00968:      .remove =   spidev_lcd_remove,
00969:
00970:      /* NOTE:  suspend/resume methods are not necessary here.
00971:       * We don't do anything except pass the requests to/from
00972:       * the underlying controller.  The refrigerator handles
00973:       * most issues; the controller driver handles the rest.
00974:       */
00975: };
00976:
00977: /*-------------------------------------------------------------------
00978:
```

Probe function will configure SPI bus and initialize TFT display.

```
00915:      spi->max_speed_hz = 19200000;
00916:      spi->mode &=~(0xf);
00917:      spi->mode |= SPI_MODE_3;
00918:      spi_setup(spi);
00919:      spi_lcd_init(spi);
00920: return status;
```

The initialization of TFT display use GPIO and SPI bus to initialize TFT controller.

```
00798: void spi_lcd_init(struct spi_device *spi)
00799: {
00800:     int i, j, n=0;
00801:     printk(KERN_WARNING"mhwld[%s][%s:%d]\n",__func__,__FILE__,__LINE__);
00802:     printk(KERN_WARNING"gpio_requst:%d\n",gpio_request(GPIO_RST,"spidev_lcd"));
00803:     printk(KERN_WARNING"gpio_requst:%d\n",gpio_request(GPIO_RS,"spidev_lcd"));
00804:     gpio_direction_output(GPIO_RST, 0);
00805:     gpio_direction_output(GPIO_RS, 0);
00806:     mdelay(50);
00807:     gpio_set_value(GPIO_RST, 1);
00808:     mdelay(50);
00809:     for (i = 0; i < ARRAY_SIZE(cmds); i++)
00810:     {
00811:         write_command(spi, cmds[i].reg_addr);
00812:         for (j = 0; j < cmds[i].len; j++)
00813:             {
00814:                 write_data(spi, spi_lcd_datas[n++]);
00815:
00816:             }
00817:         if (cmds[i].delay_ms)
00818:             mdelay(cmds[i].delay_ms);
00819:     }
00820:     clear_screen_buffer(spi,0xf800);
```

The specific initialize command is determined by TFT controller chip.

```
00746: typedef struct _spi_lcd_cmd{
00747:     unsigned char  reg_addr;
00748:     unsigned char len;
00749:     int delay_ms;
00750: }spi_lcd_cmd;
00751: spi_lcd_cmd cmds[]={
00752:         {0xCF, 3, 0},
00753:         {0xED, 4, 0},
00754:         {0xE8, 3, 0},
00755:         {0xCB, 5, 0},
00756:         {0xF7, 1, 0},
00757:         {0xEA, 2, 0},
00758:         {0xC0, 1, 0},
00759:         {0xC1, 1, 0},
00760:         {0xC5, 2, 0},
00761:         {0xC7, 1, 0},
00762:         {0x36, 1, 0},
00763:         {0x3A, 1, 0},
00764:         {0xB1, 2, 0},
00765:         {0xB6, 2, 0},
00766:         {0xF2, 1, 0},
00767:         {0x26, 1, 0},
00768:         {0xE0, 15, 0},
00769:         {0xE1, 15, 0},
00770:         {0x11, 0,  120},
00771:         {0x29, 0, 0},
```

- User space's operation to SPI display

Operation function has been provided, and only Open and Write have been offered currently.

```
00628: static const struct file_operations spidev_lcd_fops = {
00629:     .owner =    THIS_MODULE,
00630:     /* REVISIT switch to aio primitives, so that userspace
00631:      * gets more complete API coverage.  It'll simplify things
00632:      * too, except for the locking.
00633:      */
00634:     .write =    spidev_lcd_write,
00635:     .read =     spidev_lcd_read,
00636:     .unlocked_ioctl = spidev_ioctl,
00637:     .compat_ioctl = spidev_compat_ioctl,
00638:     .open =     spidev_lcd_open,
00639:     .release =  spidev_release,
00640:     .llseek =   no_llseek,
00641: };
```

Write offers function that can flash write the whole display simply.

```
00231: spidev_lcd_write(struct file *filp, const char __user *buf,
00232:         size_t count, loff_t *f_pos)
00233: {
00234:     struct spidev_data  *spidev;
00235:     ssize_t         status = 0;
00236:     unsigned long       missing;
00237:
00238:     /* chipselect only toggles at start or end of operation */
00239:     if (count > FRAMEBUFF_SIZE)
00240:         return -EMSGSIZE;
00241:
00242:     spidev = filp->private_data;
00243:
00244:     mutex_lock(&spidev->buf_lock);
00245:     missing = copy_from_user(spidev->tx_buffer, buf, count);
00246:     if (missing == 0)
00247:         status =write_data_buffer(spidev->spi,spidev->tx_buffer,count);
00248:     else
00249:         status = -EFAULT;
00250:     mutex_unlock(&spidev->buf_lock);
00251:
00252:     return status;
00253: } ? end spidev_lcd_write ?
```

5.2. Code Analysis of SPI TFT Display Driver Differences (Transplantation is Needed in This Part)

Currently, TFT LCD supported by *spidev.c* in this document includes displays of three LCD controllers: ILI9341, ILI9163 and ST7789V. If the needed LCD controller isn't included in above 3 items, transplantation can be done after reading this part.

- Pins configuration

Except 4 SPI lines, SPI LCD still needs RST (must), CD or RS (must) and BL (optional), so users need to distribute GPIO for them on platform. Here are codes:

```
0057:
0058: #define lcd_ili9341
0059: #ifdef lcd_st7789
0060: #define GPIO_RST                59
0061: #define GPIO_RS                 38
0062: #define LCD_W                   240
0063: #define LCD_H                   320
0064: #elif defined   lcd_ili9341
0065: #define GPIO_RST                1
0066: #define GPIO_RS                 38
0067: #define LCD_W                   240
0068: #define LCD_H                   320
0069: #elif defined lcd_ili9163
0070: #define GPIO_RST        24//17//LCD_CD
0071: #define GPIO_RS         42//16//LCD_RST
0072: #define GPIO_BL                 11//15//LCD_BL
0073: #define LCD_W                   160
0074: #define LCD_H                   128
0075: #endif
```

First, users need to select LCD model, such as *lcd_ili9341*, please simulate definition if the LCD model isn't included in above 3 items. Then, configure GPIO number for GPIO_RST, GPIO_RS and GPIO_BL according to hardware connection, at last, configure LCD's width and height.

- Configuration of command and data's read-write sequence (Usually Must)

During the communication of data and command with LCD controller, data lines are SPI data lines, CD or RS is used to indicate the content in transmission is data or command. So, users need to define the status of GPIO_RS of three functions as shown in following picture. If LCD controller requests the RS be low when writing command, then make it by *gpio_direction_output(GPIO_RS,0)*; If requests high, then by *gpio_direction_output(GPIO_RS,1)*. The three functions in the following picture need to be configured, but usually LCD controller configuration are the same, so users needn't to modify it at first if not clear.

```
00650:
00651: void write_command(struct spi_device *spi, unsigned char cmd)
00652: {
00653:
00654:    // dc , command:0
00655:    gpio_direction_output(GPIO_RS, 0);
00656:    spi_write(spi, &cmd,1);
00657: }
00658:
00659: void write_data(struct spi_device *spi,unsigned char data )
00660: {
00661:    // dc , data:1
00662:    gpio_direction_output(GPIO_RS, 1);
00663:    spi_write(spi,&data,1);
00664: }
00665:
00666: void read_data(struct spi_device *spi,unsigned char *data,char len )
00667: {
00668:    // dc , data:1
00669:    gpio_direction_output(GPIO_RS, 1);
00670:    spi_read(spi,data,len);
00671: }
00672:
00673:
```

- Configuration of LCD display function (Usually Not Must)

The following three functions are needed during LCD displaying. LCD controller is different, but the operations of some controllers are the same, even command code. Users need to rewrite the function based one operation sequence of selected LCD controller, usually the rewriting is needn't.

```
00682: void addr_set(struct spi_device *spi, unsigned int x1,unsigned int y1,unsigned int x2,unsigned int y2)
00683: {
00684:     write_command(spi, 0x2a);
00685:     write_data(spi, x1>>8);
00686:     write_data(spi, x1&0xff);
00687:         write_data(spi, x2>>8);
00688:     write_data(spi, x2&0xff);
00689:
00690:     write_command(spi, 0x2b);
00691:     write_data(spi, y1>>8);
00692:     write_data(spi, y1&0xff);
00693:         write_data(spi, y2>>8);
00694:     write_data(spi, y2&0xff);
00695:
00696:     write_command(spi, 0x2c);
00697: }
00698:
00699:
00700: void clear_screen(struct spi_device *spi,unsigned short color)
00701: {
00702:     unsigned short i,j;
00703:     unsigned char vh,vl;
00704:     vh=color>>8;
00705:     vl=color;
00706:     addr_set(spi,0,0,LCD_W-1,LCD_H-1);
00707:     for(i=0;i<LCD_H;i++)
00708:     {
00709:         for(j=0;j<LCD_W;j++)
00710:         {
00711:             write_data(spi,vh);
00712:             write_data(spi,vl);
00713:         }
00714:     }
00715: }
00716:
```

```
00717: void clear_screen_buffer(struct spi_device *spi,unsigned short color)
00718: {
00719:     struct spidev_data  *spidev_lcd;
00720:     unsigned short i,j;
00721:     unsigned char *buf;
00722:     char init=0;
00723:     spidev_lcd=(struct spidev_data      *)spi->dev.driver_data;
00724:     if(spidev_lcd!=0)
00725:         buf=spidev_lcd->tx_buffer;
00726:     else
00727:     {
00728:         init=1;
00729:         buf=kmalloc(FRAMEBUFF_SIZE, GFP_KERNEL);
00730:     }
00731:     unsigned char vh,vl;
00732:     vh=color>>8;
00733:     vl=color;
00734:     addr_set(spi,0,0,LCD_W-1,LCD_H-1);
00735:     for(i=0;i<=LCD_H;i++)
00736:     {
00737:         for(j=0;j<=LCD_W;j++)
00738:         {
00739:             buf[(i*LCD_W+j)*2]=vh;
00740:             buf[(i*LCD_W+j)*2+1]=vl;
00741:         }
00742:     }
00743:     write_data_buffer(spi,buf,FRAMEBUFF_SIZE);
00744:     if(init==1)
00745:     {
00746:         kfree(buf);
00747:         init=0;
00748:     }
00749: }« end clear_screen_buffer »
00750:
```

- Configuration of LCD Controller Initialization (Usually Must)

The initialization procedure of various LCD controllers is different. Data of *cmds* and *spi_lcd_datas* as shown in following picture need to be written according to LCD features. The first data in *cmds* means command code, the second data means the number of the data that send after sending the command code (please write 0 if no need to send data), the third data means waiting time in ms after sending the command code and the data mentioned in second data's definition (please write 0 if no delay); *spi_lcd_datas* is where to store data in the order of command codes after sending command codes.

```
00750:
00751: typedef struct _spi_lcd_cmd{
00752:     unsigned char reg_addr;
00753:     unsigned char len;
00754:     int delay_ms;
00755: }spi_lcd_cmd;
00756:
00757: #ifdef lcd_ili9341
00758:
00759:
00760: spi_lcd_cmd cmds[]={
00761:         {0xCF, 3, 0},
00762:         {0xED, 4, 0},
00763:         {0xE8, 3, 0},
00764:         {0xCB, 5, 0},
00765:         {0xF7, 1, 0},
00766:         {0xEA, 2, 0},
00767:         {0xC0, 1, 0},
00768:         {0xC1, 1, 0},
00769:         {0xC5, 2, 0},
00770:         {0xC7, 1, 0},
00771:         {0x36, 1, 0},
00772:         {0x3A, 1, 0},
00773:         {0xB1, 2, 0},
00774:         {0xB6, 2, 0},
00775:         {0xF2, 1, 0},
00776:         {0x26, 1, 0},
00777:         {0xE0, 15, 0},
00778:         {0xE1, 15, 0},
00779:         {0x11, 0,  120},
00780:         {0x29, 0, 0},
00781: };
00782:
```

```
782:
783: unsigned char spi_lcd_datas[] = {
784:     0x00, 0xd9, 0x30,                   // command: 0xCF
785:     0x64, 0x03, 0x12,0x81,              // command: 0xED
786:     0x85, 0x10, 0x78,                    // command: 0xE8
787:     0x39, 0x2c, 0x00, 0x34, 0x02,       // command: 0xCB
788:     0x20,                               // command: 0xF7
789:     0x00, 0x00,                         // command: 0xEA
790:     0x21,                               // command: 0xC0
791:     0x12,                               // command: 0xC1
792:     0x32, 0x3C,                         // command: 0xC5
793:     0xC1,                               // command: 0xC7
794:     0x08,                               // command: 0x36
795:     0x55,                                // command: 0x3A
796:     0x00, 0x18,                         // command: 0xB1
797:     0x0A, 0xA2,                         // command: 0xB6
798:     0x00,                               // command: 0xF2
799:     0x01,                               // command: 0x26
800:     0x0F, 0x20, 0x1E, 0x09, 0x12, 0x0B, 0x50, 0xBA, 0x44, 0x09, 0x14, 0x05, 0x23, 0x21, 0x00, //command: 0xE0
801:     0x00, 0x19, 0x19, 0x00, 0x12, 0x07, 0x2D, 0x28, 0x3F, 0x02, 0x0A, 0x08, 0x25, 0x2D, 0x0F, //command: 0xE1
802:
803: };
```

- Editing LCD Pin Initialization Function (Usually Must)

Users need to request GPIO in driver with *gpio_request*, and then restart operation sequence according to LCD actual situation, backlight will also be opened through this request if needed.

```
..
5: void lcd_gpio_init(void)
6: {
7:         printk(KERN_WARNING"gpio_requst:%d\n",gpio_request(GPIO_RST,"spidev_lcd"));
8:         printk(KERN_WARNING"gpio_requst:%d\n",gpio_request(GPIO_RS,"spidev_lcd"));
9:         gpio_direction_output(GPIO_RST, 0);
0:         gpio_direction_output(GPIO_RS, 0);
1:         mdelay(50);
2:         gpio_set_value(GPIO_RST, 1);
3:         mdelay(50);
4: }
5:
```

6 Testing of Refresh Picture


Image2Lcd.exe


1.jpg

Please install , then open the software and select the picture , please check the parameters as following:

Please be noted that the software Image2lcd.exe mentioned above only has Chinese version, users can download other similar software in English to do above steps.

Click “Save” to generate *1.bin*, then upload *1.bin* to file system of the main board, and enter following to console of the main board:

```
#dd if=./1.bin of=/dev/spidev6.0
```

The display will show the following:

6.1. FAQ

- The selected GPIO number has been used for other purpose, such as interrupts, which block command and data's communication.

Please check Kernel log (*dmesg*) to make sure the GPIO has been registered successfully, if not, users can re-select GPIO or remove other functions that occupy this GPIO.

- Failed to write SPI

Please check Kernel log (*dmesg*), there should be an abnormal in SPI speed. Usually, the SPI speed of this driver is faster than it of DTS, please set the SPI speed of this driver as maximum speed of DTS.

7 Appendix A References

Table 1: Terms and Abbreviations

Abbreviation	Description
SPI	Serial Peripheral Interface
LCD	Liquid Crystal Display
LTE	Long Term Evolution
SDK	Software Development Kit
PCB	Printed Circuit Board
TFT	Thin Film Transistor
CPU	Central Processing Unit
DMA	Direct Memory Access
API	Application Programming Interface
MTD	Memory Technology Device
TTY	Teletypes
FB	Framebuffer
GPU	Graphics Processing Unit
AD	Analog-to-Digital
GPIO	General-purpose input/output
GND	Ground
VCC	Volt Current Condenser
LED	Light Emitting Diode
REST	Reset

D/C	Down Convertor
CLK	Clock
SDI	Serial Digital Interface
MOSI	Master Output/Slave Input
SDO	Service Data Objects
DTS	Digital Theater Systems
UART	Universal Asynchronous Receiver/Transmitter