**实验四. LCD 驱动及控制实验**

1. 实验目的

 了解 LCD 基本概念与原理

 了解 Linux 下 LCD 的 Framebuffer 结构原理

 了解用总线方式驱动 LCD 模块

 熟悉用 ARM 内置的LCD 控制器驱动

2. 实验内容

 学习 LCD 显示器的基本原理，了解 Linux 下 Framebuffer 结构原理

 了解用总线方式驱动 LCD 模块，并编程实现 LCD 显示

3. 实验环境

 硬件：IMX6 教学平台，PC 机酷睿 i3 以上, 硬盘 120G 以上, 内存 2G 以上

 软件：Vmware Workstation +Yocto 项目

4. 实验原理

**4.1 硬件接口原理**

u **LCD（Liquid** **Crystal** **Display）原理**

液晶得名于其物理特性：它的分子晶体，以液态存在而非固态。这些晶体分子的液体特性使得它具有 两种非常有用的特点：

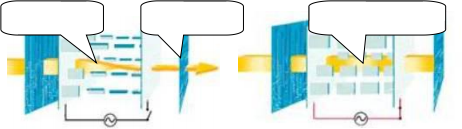
1 、如果让电流通过液晶层，这些分子将会以电流的流向方向进行排列，如果没有电流，它们将会彼此 平行排列。

2 、如果提供了带有细小沟槽的外层，将液晶倒入后，液晶分子会顺着槽排列，并且内层与外层以同样 的方式进行排列。

液晶的第三个特性是很神奇的：液晶层能使光线发生扭转。液晶层表现的有些类似偏光器，这就意味 着它能够过滤除了那些从特殊方向射入之外的所有光线。此外，如果液晶层发生了扭转，光线将会随之扭 转，以不同的方向从另外一个面中射出。

液晶的这些特点使得它可以被用来当作一种开关——即可以阻碍光线，也可以允许光线通过。液晶单 元的底层是由细小的脊构成的，这些脊的作用是让分子呈平行排列。上表面也是如此，在这两侧之间的分 子平行排列，不过当上下两个表面之间呈一定的角度时，液晶随着两个不同方向的表面进行排列，就会发 生扭曲。结果便是这个扭曲的螺旋层使通过的光线也发生扭曲。如果电流通过液晶，所有的分子将会按照 电流的方向进行排列，这样就会消除光线的扭转。如图 3.4.1 所示，如果将一个偏振滤光器放置在液晶层的

上表面，扭转的光线通过(如 A)，而没有发生扭转的光线(如 B)将被阻碍。因此可以通过电流的通断改变 LCD 中的液晶排列，使光线在加电时射出，而不加电时被阻断。也有某些设计为了省电的需要，有电流时，光 线不能通过，没有电流时，光线通过。



扭转的光线 偏振滤光器 没有扭转的光线

A 加电时 B 不加电时

图 3.4. 1 LCD 原理

LCD 显示器的基本原理就是通过给不同的液晶单元供电，控制其光线的通过与否，从而达到显示的目 的。因此，LCD 的驱动控制归于对每个液晶单元的通断电的控制，每个液晶单元都对应着一个电极，对其 通电，便可使光线通过（也有刚好相反的，即不通电时光线通过，通电时光线不通过）。

u **电致发光**

LCD 的发光原理是通过控制加电与否来使光线通过或挡住，从而显示图形。光源的提供方式有两种： 透射式和反射式。笔记本电脑的 LCD 显示屏即为透射式，屏后面有一个光源，因此外界环境可以不需要光 源。而一般微控制器上使用的 LCD 为反射式，需要外界提供光源，靠反射光来工作。电致发光（EL）是液 晶屏提供光源的一种方式。电致发光的特点是低功耗，与二极管发光比较而言体积小。

电致发光（EL）是将电能直接转换为光能的一种发光现象。电致发光片是利用此原理经过加工制作而 成的一种发光薄片，如图 3.4.2 所示。其特点是：超薄、高亮度、高效率、低功耗、低热量、可弯曲、抗冲 击、长寿命、多种颜色选择等。因此，电致发光片被广泛应用于各种领域。

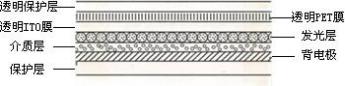


图 3.4.2 电致发光片的基本结构

u **LCD** **的驱动控制**

市面上出售的 LCD 有两种类型：

一种是带有驱动电路的 LCD 显示模块，这种 LCD 可以方便地与各种低档单片机进行接口，如 8051 系 列单片机，但是由于硬件驱动电路的存在，体积比较大。这种模式常常使用总线方式来驱动。

另一种是 LCD 显示屏，没有驱动电路，需要与驱动电路配合使用，如图 3.4.3 所示。特点是体积小， 但却需要另外的驱动芯片。也可以使用带有LCD 驱动能力的高档MCU 驱动，如ARM 系列的EXYNOS 4412、 PXA270 等。

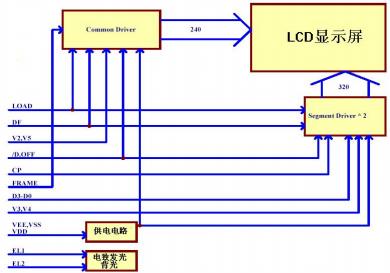
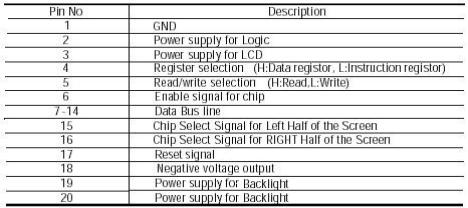


图 3.4.3 不带驱动电路的 LCD 结构

( 1) 总线驱动方式

一般带有驱动模块的 LCD 显示屏使用这种驱动方式， 由于 LCD 已经带有驱动硬件电路，因此模块给 出的是总线接口，便于与单片机的总线进行接口。驱动模块具有八位数据总线，外加一些电源接口和控制 信号。而且自带显示缓存，只需要将要显示的内容送到显示缓存中就可以实现内容的显示。 由于只有八条 数据线，因此常常通过引脚信号来实现地址与数据线复用，以达到把相应数据送到相应显示缓存的目的。 表 3.4.1 为一个典型的显示模块（HY－12864B）提供的总线接口。

**表** **3.4.1 典型带驱动液晶模块的总线接口**



(2) 控制器扫描方式

IMX6 中具有内置的LCD 控制器，它具有将显示缓存（在系统存储器中）中的 LCD 图象数据传输到外 部 LCD 驱动电路的逻辑功能。支持 DSTN（被动矩阵或叫无源矩阵）和 TFT（主动矩阵或叫有源矩阵）两 种 LCD 屏，并支持黑白和彩色显示。对于不同尺寸的 LCD ，具有不同数量的垂直和水平象素、数据接口的 数据宽度、接口时间及刷新率，而 LCD 控制器可以进行编程控制相应的寄存器值，以适应不同的 LCD 显 示板。

（3）IMX6 型平台LCD 部分硬件电路图



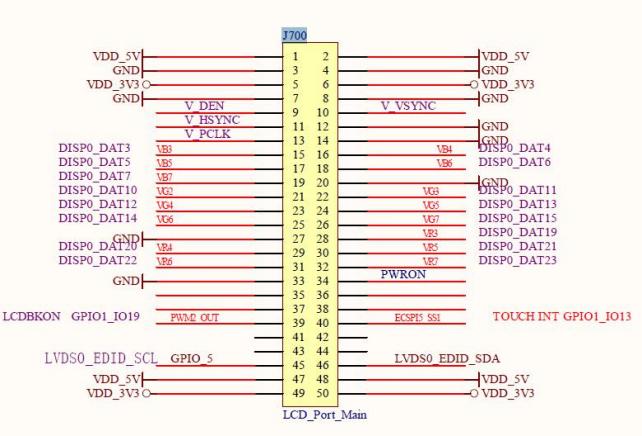


图 3.4.4 LCD 的硬件电路图

**4.2 软件接口介绍**

u **Linux** **的帧缓冲设备**

帧缓冲（framebuffer）是 Linux 为显示设备提供的一个接口，把显存抽象后的一种设备，他允许上层 应用程序在图形模式下直接对显示缓冲区进行读写操作。这种操作是抽象的，统一的。用户不必关心物理 显存的位置、换页机制等等具体细节。这些都是由Framebuffer 设备驱动来完成的。帧缓冲驱动的应用广泛， 在 linux 的桌面系统中，Xwindow 服务器就是利用帧缓冲进行窗口的绘制。尤其是通过帧缓冲可显示汉字 点阵，成为 Linux 汉化的唯一可行方案。

帧缓冲设备对应的设备文件为/dev/fb\* ，如果系统有多个显示卡，Linux 下还可支持多个帧缓冲设备， 最多可达 32 个，分别为/dev/fb0 到/dev/fb31 ，而/dev/fb 则为当前缺省的帧缓冲设备，通常指向/dev/fb0 。当 然在嵌入式系统中支持一个显示设备就够了。帧缓冲设备为标准字符设备，主设备号为 29 ，次设备号则从 0 到 31 。分别对应/dev/fb0 到/dev/fb31。

u **FrameBuffer** **在** **Linux** **中的实现和机制**

Framebuffer 对应的源文件在内核源码的/drivers/video/目录下。总的抽象设备文件为 fbcon.c ，在这个目

录下还有与各种显示设备驱动相关的源文件。

FrameBuffer 设备驱动基于如下两个文件：

include/linux/fb.h

drivers/video/fbmem.c

1 、在 fb.h 中定义了 framebuffer 所使用的重要数据结构：

1).Struct fb\_var\_screeninfo 描述显卡的特性的。通常是被用户设置的。 2).Struct fb\_fix\_screeninfo 定义了显卡的硬件特性，是不能改变的。

3).Struct fb\_cmap 描述设备无关的颜色映射信息。可以通过 FBIOGETCMAP 和 FBIOPUTCMAP 对应的 ioctl 操作设定或获取颜色映射信息.

4).Struct fb\_info 定义了当前显卡 framebuffer 设备状态，一个显卡可能有两个 framebuffer ，在这种情况 下，就需要两个fb\_info 结构。这个结构是唯一在内核空间可见的。在这个结构中有一个 fb\_ops 指针，指向 驱动设备工作所需的函数集。

5).struct fb\_ops 用户应用可以使用 ioctl()系统调用来操作设备，这个结构就是用以支持 ioctl()的这些操作 的。（注: fb\_ops 结构与 file\_operations 结构不同，fb\_ops 是底层操作的抽象,而 file\_operations 是提供给上 层系统调用的接口，可以直接调用. ）ioctl()系统调用在文件 fbmem.c 中实现，通过观察可以发现 ioctl()命令 与 fb\_ops’s 中函数的关系:

FBIOGET\_VSCREENINFO fb\_get\_var FBIOPUT\_VSCREENINFO fb\_set\_var FBIOGET\_FSCREENINFO fb\_get\_fix FBIOPUTCMAP fb\_set\_cmap

FBIOGETCMAP fb\_get\_cmap

FBIOPAN\_DISPLAY fb\_pan\_display

如果我们定义了fb\_XXX\_XXX 方法，用户程序就可以使用 FBIOXXXX 宏的 ioctl()操作来操作硬件

2 、 fbmem.c 处于 Framebuffer 设备驱动技术的中心位置.它为上层应用程序提供系统调用也为下一层的 特定硬件驱动提供接口；那些底层硬件驱动需要用到这儿的接口来向系统内核注册它们自己. fbmem.c 为所 有支持FrameBuffer 的设备驱动提供了通用的接口。

1) 全局变量

struct fb\_info \*registered\_fb[FB\_MAX]; int num\_registered\_fb;

这两变量记录了所有 fb\_info 结构的实例，fb\_info 结构描述显卡的当前状态，所有设备对应的 fb\_info 结构都保存在这个数组中，当一个 FrameBuffer 设备驱动向系统注册自己时，其对应的 fb\_info 结构就会添加到这个结构中，同时 num\_registered\_fb 为自动加 1.

2）fbmem.c 实现了如下函数.

register\_framebuffer(struct fb\_info \*fb\_info); unregister\_framebuffer(struct fb\_info \*fb\_info);

这两个是提供给下层 FrameBuffer 设备驱动的接口，设备驱动通过这两函数向系统注册或注销自己。几 乎底层设备驱动所要做的所有事情就是填充fb\_info 结构然后向系统注册或注销它。

u **应用程序端：**

对/dev/fb 设备的操作，应用程序的操作主要有这几种：

1） 读/写（read/write）/dev/fb：相当于读/写屏幕缓冲区。 例如：

用 cp /dev/fb0 tmp 命令可将当前屏幕的内容拷贝到一个文件中，而命令 cp tmp > /dev/fb0 则将图形文 件 tmp 显示在屏幕上。

2） 映射（map）操作：由于 Linux 工作在保护模式，每个应用程序都有自己的虚拟地址空间，在应用 程序中是不能直接访问物理缓冲区地址的。为此，Linux 在文件操作 file\_operations 结构中提供了 mmap 函 数，可将文件的内容映射到用户空间。对于帧缓冲设备，则可通过映射操作，可将屏幕缓冲区的物理地址 映射到用户空间的一段虚拟地址中，之后用户就可以通过读写这段虚拟地址访问屏幕缓冲区，在屏幕上绘 图了。

例如：

mmap 用法：

#include <sys/mman.h>

void \*mmap (void \*start, size\_t length, int prot, int flags,int fd, off\_t

offset);

int munmap (void \*start, size\_t length);

参数：

start： 映射区的开始地址。 length：映射区的长度。

prot： 期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过 or 运算合理 地组合在一起

PROT\_EXEC :页内容可以被执行 PROT\_READ :页内容可以被读取 PROT\_WRITE :页可以被写入

PROT\_NONE :页不可访问

flags： 指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的

组合体

MAP\_FIXED :使用指定的映射起始地址，如果由 start 和 len 参数指定的内存区重叠于现存的 映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起始地址 必须落在页的边界上。

MAP\_SHARED :与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于

输出到文件。直到msync()或者 munmap()被调用，文件实际上不会被更新。

MAP\_PRIVATE :建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这 个标志和以上标志是互斥的，只能使用其中一个。

MAP\_NORESERVE :不要为这个映射保留交换空间。当交换空间被保留，对映射区修改的可能

会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。 MAP\_LOCKED :锁定映射区的页面，从而防止页面被交换出内存。

MAP\_GROWSDOWN :用于堆栈，告诉内核 VM 系统，映射区可以向下扩展。 MAP\_ANONYMOUS :匿名映射，映射区不与任何文件关联。

MAP\_ANON :MAP\_ANONYMOUS 的别称，不再被使用。

MAP\_FILE :兼容标志，被忽略。

MAP\_32BIT :将映射区放在进程地址空间的低 2GB ，MAP\_FIXED 指定时会被忽略。当前这 个标志只在 x86-64 平台上得到支持。

MAP\_POPULATE :为文件映射通过预读的方式准备好页表。随后对映射区的访问不会被页违

例阻塞。

MAP\_NONBLOCK :仅和 MAP\_POPULATE 一起使用时才有意义。不执行预读，只为已存在于 内存中的页面建立页表入口。

fd：有效的文件描述词。如果 MAP\_ANONYMOUS 被设定，为了兼容问题，其值应为-1。

offset：被映射对象内容的起点。 返回说明：

成功执行时，mmap()返回被映射区的指针，munmap()返回 0 。失败时，mmap()返回 MAP\_FAILED[其值 为(void \*)-1] ，munmap 返回-1 。errno 被设为以下的某个值

EACCES：访问出错

EAGAIN：文件已被锁定，或者太多的内存已被锁定 EBADF：fd 不是有效的文件描述词

EINVAL：一个或者多个参数无效

ENFILE： 已达到系统对打开文件的限制

ENODEV：指定文件所在的文件系统不支持内存映射

ENOMEM： 内存不足，或者进程已超出最大内存映射数量 EPERM： 权能不足，操作不允许

ETXTBSY： 已写的方式打开文件，同时指定 MAP\_DENYWRITE 标志 SIGSEGV：试着向只读区写入

SIGBUS：试着访问不属于进程的内存区

3） I/O 控制：对于帧缓冲设备，对设备文件的 ioctl 操作可读取/设置显示设备及屏幕的参数，如 分辨率，显示颜色数，屏幕大小等等。ioctl 的操作是由底层的驱动程序来完成的。

例如：

ioctl(fbfd, FBIOGET\_FSCREENINFO, &finfo)

获取 fb\_var\_screeninfo 结构的信息，在 linux/include/linux/fb.h 定义。 ioctl(fbfd, FBIOGET\_VSCREENINFO, &vinfo)

获取 fb\_fix\_screeninfon 结构的信息。在 linux/include/linux/fb.h 定义。

在应用程序中，操作/dev/fb 的一般步骤如下：

1 ．打开/dev/fb 设备文件。

2 ．用 ioctl 操作取得当前显示屏幕的参数，如屏幕分辨率，每个像素点的比特数。根据屏幕参数可计 算屏幕缓冲区的大小。

3 ．将屏幕缓冲区映射到用户空间（mmap）。

4 ．映射后就可以直接读写屏幕缓冲区，进行绘图和图片显示了。 典型程序段如下：

#include <linux/fb.h>

int main ()

{

int fb = 0; void \*fb mem;

\_

struct fb var screeninfo vinfo;

\_ \_

struct fb fix screeninfo finfo;

long int crensize = 0;

/\*打开设备文件\*/

fb = open ("/dev/fb0", O\_RDWR); /\*取得屏幕相关参数\*/

ioctl (fb, FBIOGET\_FSCREENINFO, &finfo);

ioctl (fb, FBIOGET\_VSCREENINFO, &vinfo); /\*计算屏幕缓冲区大小\*/

screensize = vinfo.xres \* vinfo .yres \* vinfo.bits\_per\_pixel / 8; /\*映射屏幕缓冲区到用户地址空间\*/

fb\_mem= (char\*)mmap (0,screensize,PROT\_READ |PROT\_WRITE,MAP\_SHARED, fb, 0); /\*下面可通过 fbp 指针读写缓冲区\*/

……

/\*释放缓冲区，关闭设备\*/

munmap (fb\_mem, screensize);

close (fb);

}

**4.3 关键代码分析**

本次实验使用的应用程序主要是通过 lcd.c 源文件来与系统底层驱动程序交互的，在 lcd.c 中完成了 framebuffer 设备的初始化以及通过映射地址后对 ioctrl 接口的设置来完成用户空间的调用。

#include "lcd.h"

int current vt;

struct termios term;

// Framebuffer device routine

int fb\_con = 0;

void \* frame\_base = 0;

/\* framebuffer device handle \*/

/\* lcd framebuffer 用户空间地址指针 \*/

// Framebuffer 初始化函数 .

// Failed return -1, succeed return 0.

int fb\_Init (void)

{

struct fb fix screenin fo finfo; /\* fixed screen information \*/

\_ \_

struct fb var screeninfo vinfo; /\* variable screen information

\_ \_

\*/

struct termios current;//定义终端对象 //定义 RGB buffer 初始化 palette 调色板

unsigned short red [256], green [256], blue [256];

struct fb\_cmap new\_map = {0, 256, red, green, blue, NULL}; /\*

new system palette \*/

//获取输入终端属性，保存预留 tcgetattr (0, &term);

current = term;

//设置输入终端属性

current.c\_lflag &= ~~ICANON;

current.c\_lflag &= ECHO;

current.c\_cc [VMIN] = 1;

current.c\_cc [VTIME] = 0;

//设置终端属性立即生效

tcsetattr (0, TCSANOW, &current);

// 打开虚拟控制台终端，设置为图形模式，防止终端刷新 current\_vt = open ("/dev/tty", O\_RDWR);

ioctl (current\_vt, KDSETMODE, KD\_GRAPHICS); // 打开 framebuffer LCD 显示设备

fb\_con = open ("/dev/fb0", O\_RDWR, 0);

if (fb\_con < 0)

{

printf ("Can 't open /dev/fb0.\n");

return -1;

}

// 获取 显示设备硬件的 fixed 结构信息

if (ioctl (fb\_con, FBIOGET\_FSCREENINFO, &finfo) < 0)

{

printf ("Can 't get FSCREENINFO.\n");

close (fb con);



return -1;

}

// 获取 variable 显示信息

if (ioctl (fb\_con, FBIOGET\_VSCREENINFO, &vinfo) < 0)

{

printf ("Can 't get VSCREENINFO.\n");

close (fb con);

return -1;

}

// 使用系统时间初始化随机数发生器种子 srand (time (0));

// 设置新的调色板

fb MakePalette (&new\_map);

\_



// 应用 framebuffer 颜色

if (finfo .visual == FB\_VISUAL\_DIRECTCOLOR || vinfo .bits\_per\_pixel ==

8)

{

if (ioctl (fb\_con, FBIOPUTCMAP, &new\_map) < 0) {

printf ("Error putting Colormap .\n"); return -1;

} }



// 配置 framebuffer 颜色像素深度 switch (vinfo.bits\_per\_pixel) {

case 8:

fb\_pixel\_size = 1;

break;



case 16:

fb\_pixel\_size = 2;

vinfo.red.offset = 11;

vinfo.red.length = 5;

vinfo.green.offset = 5;

vinfo.green.length = 6;

vinfo.blue .offset = 0;

vinfo.blue .length = 5;

break; case 32:

fb\_pixel\_size = 4;

vinfo.red.offset = 24;

vinfo.red.length = 8;

vinfo.green.offset = 16;

vinfo.green.length = 8;

vinfo.blue .offset = 8;

vinfo.transp .offset = 0;

vinfo.blue .length = 8;

vinfo.transp.length = 8;

break;



default:

fprintf (stderr, "Current color depth is NOT

surpported .\n");

fb\_pixel\_size = 1;

break;

}

// 设置 framebuffer vinfo 结构

if (ioctl (fb\_con, FBIOPUT\_VSCREENINFO, &vinfo) < 0)

{

fprintf (stderr, "Couldn 't set ideal mode at

FBIOPUT VSCREENINFO");

return -1;

}

// 获取屏幕信息，并保存

if (ioctl (fb\_con, FBIOGET\_VSCREENINFO, &vinfo) < 0)

{

fprintf (stderr, "ioctl FBIOGET\_VSCREENINFO");

return -1;

}

fb\_width = vinfo .xres\_virtual;

fb\_height = vinfo.yres\_virtual/4;

fb\_bpp = vinfo .bits\_per\_pixel;

fb\_line\_size = finfo .line\_length;

fb\_buffer\_size = finfo .smem\_len;

// 映射 fb\_con 结构 buffer 到用户空间的一段内存中，获取映射后地址指针

frame\_base = mmap (NULL, BUFFER\_SIZE, PROT\_READ |PROT\_WRITE, MAP\_SHARED, fb\_con, 0);

if (frame\_base == MAP\_FAILED)

{

fprintf (stderr, "Can 't mmap.\n");

close (fb con);

return -1;

}

printf ("\nVideo memory address = 0x%x\n",frame\_base);

printf ("Video visible resolution : x\_res = %ld, y\_res = %ld\n",

SCREEN\_WIDTH, SCREEN\_HEIGHT);

// 清屏

fb Clear (SYS BLACK);

return 0;

}

test.c 源文件主要包含了 lcd.c 函数中对 framebuffer 结构的接口进行封装后的图形显示接口，篇幅所限， 其他接口用户可以通过阅读实验源码自行分析。

5. 实验步骤

**5.1 实验目录**

/home/uptech/fsl-6dl-source/kernel-4.9.88//IMX6/SRC/exp/lcd/

1 、进入内核目录,在内核中添加 framebuffer 设备支持：

root@uptech-virtual-machine:/home/uptech#cd fsl-6dl-source/kernel-4.9.88

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#export ARCH=arm

[root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#](mailto:root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-3.14.28#)make menuconfig

2 、配置内核对 framebuffer 的相关支持

Device Drivers --->

Graphics support --->

Frame buffer Devices --->

驱动配置如下图：

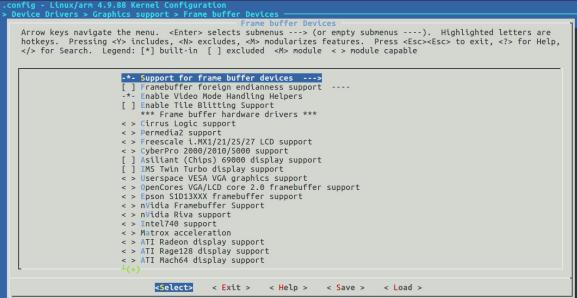


图 3.4.5 内核配置

退出保存设置。

3 、项目编译

[root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#](mailto:root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-3.14.28#) make z Image root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88# make

imx6dl-sabresd.dtb

编译成功后会在源码目录的 arm/arch/boot/目录下生成内核压缩文件 zImage 、在 arch/arm/boot 下的 IMX6DL-sabresd.dtb 文件。

4 、按照 IMX6 光盘配套烧写文档将新生成的内核镜像文件烧写到 IMX6 设备中，这里不在赘述。

**备注：以上在内核中添加对** **framebuffer 设备的支持的步骤，在** **IMX6 设备出厂自带内核中已经默认添** **加进来了，用户可以省略以上步骤。以上步骤在于重现系统的构造。**

**5.2 编译源程序**

1 、进入宿主机实验目录：

root@uptech-virtual-machine:~# cd /IMX6/SRC/exp/driver/04\_lcd/

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/04\_lcd# ls

Makefile ascii.lib driver hz16 lcd.c lcd.h lcd.o test.c test.o test lcd

\_

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/04\_lcd# source

/opt/fsl-imx-wayland/4.9.88-2.0.0/environment-setup-cortexa9hf-neon-poky-linux-gnueabi

2 、清除中间代码，重新编译

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/04\_lcd# make clean rm -f test lcd \*.o

\_

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| root@uptech-virtual-machine:/IMX6/SRC/exp/driver/04\_lcd# make | | | | |
| rm -f test lcd \*.o | | | | |
| arm-linux-gcc -O2 -c -o test.o test.c | | | | |
| arm-linux-gcc | -O2 | -c -o | lcd.o lcd.c |  |
| arm-linux-gcc | -O2 | -o | test lcd test.o | lcd.o |
| arm-linux-strip test\_lcd | | | | |
| root@uptech-virtual-machine:/IMX6/SRC/exp/driver/04\_lcd# ls | | | | |
| Makefile ascii.lib driver hz16 lcd.c lcd.h lcd.o test.c test.o test lcd | | | | |
| root@uptech-virtual-machine:/IMX6/SRC/exp/driver/04\_lcd# | | | | |
| 当前目录下生成可执行程序 test\_lcd。  **5.3 NFS 挂载实验目录测试**  1 、 启动 IMX6 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。 | | | | |
| root@IMX6DLsabresd:~# mount -t nfs 192.168.12.157:/IMX6 /mnt/ | | | | |
| 2 、 进入串口终端的 NFS 共享实验目录。 | | | | |
| root@IMX6DLsabresd:~# cd /mnt/nfs/SRC/exp/driver/04\_lcd | | | | |
| root@IMX6DLsabresd:/mnt/nfs/SRC/exp/driver/04 lcd# ls | | | | |
| Makefile ascii.lib driver hz16 lcd.c lcd.h lcd.o test.c test.o | | | | |
| root@IMX6DLsabresd:/mnt/nfs/SRC/exp/driver/04\_lcd# | | | | |
| 3 、 执行 test\_lcd 应用程序测试 framebuffer 驱动。 | | | | |
| root@IMX6DLsabresd:/mnt/nfs/SRC/exp/driver/04\_lcd# ./test\_lcd  Press any key to continue  ... | | | | |
| 4 、 此时按下 PC 机端回车键，可以观察 UP-IMX6DL 设备 LCD 显示输出变化。  6. 实验效果    图 3.4.8 实验效果 | | | | |