

# 集中器参考设计软件手册

## 1 概述

飞思卡尔集中器参考设计是根据中国国家电网公司文件-《电力用户用电信息采集系统功能规范》附件 Q/GDW 375.2-2009《电力用户用电信息采集系统型式规范：集中器形式规范》的定义进行的软件及硬件平台设计。主要为电力系统集中器设备设计供应商提供给予飞思卡尔高端 ColdFire 芯片处理器的平台化设计，以简化并加速设计者的设计流程，降低研发风险及成本，缩短研发时间，使设计者可以迅速的基于该平台开发出自己的集中器设备产品。本软件手册的所有说明是基于飞思卡尔集中器参考硬件平台，请参考集中器参考平台硬件手册阅读本手册。

## 2 参考设计软件

参考设计的软件包括：

- 交叉编译环境及 2.6.29 标准 Linux 内核，Jffs2 文件系统，支持网络文件系统调试
- U-boot 启动 bootloader，带 tftp 远程下载，支持网络文件系统调试，可以从 SPI 启动或者 Nand Flash 启动
- LCD 模块驱动以及中英文字库
- 键盘中断扫描驱动
- 红外模块驱动
- 双以太网驱动，支持 NFS，web server 等
- USB 驱动，支持 Mass Storage 类设备
- 内部及外部 RTC 驱动
- SPI 总线驱动
- I2C 总线驱动
- 内部看门狗驱动，外部看门狗驱动将在下一版发布
- 串口驱动
- EDMA 驱动
- 一个测试的 DEMO 应用程序

其他的一些驱动在集中器方案中不会用到但也提供：

- 高精度定时器驱动
- CAN 总线设备驱动
- eSDHC 驱动
- CAU 驱动

## 2.1 Freescale BSP 环境介绍

飞思卡尔发布的 linux BSP 是 LTIB(Linux Target Image Builder)安装镜像。LTIB 可运行在安装有 linux 的 x86 PC 上。

LTIB 需要以下硬件或软件支持:

- 网卡
- 串口
- 1GB 可用硬盘空间
- NFS 服务
- TFTP 服务
- rsync 和 perl

注意: 不是所有的 linux 操作系统上都可以正常使用 LTIB, 飞思卡尔测试过以下 linux 系统, 可正常运行 LTIB:

- Redhat: 7.3, 8.0, 9.0
- Fedora Core: 1, 2, 3
- Debian: 3.1r0 (stable), unstable
- SuSE: 8.2, 9.2, 10.0, 10.2

目前本 LTIB 镜像中提供的 BSP 组件有: Linux 2.6.29 内核、10 个串口支持、NAND MTD 设备支持、TCP/IP 协议栈和 2 个以太网口支持、DMA Timer、eDMA、Flex CAN、DSPI、I2C、USB Host 和 OTG、NFS 和 jffs2 文件系统支持、u-boot (Flexbus、SBF 和 NFC 启动)、GNU gcc 4.4.1, eglibc 2.10.54, binutils-2.19.51 和 elf2flt。

### ● 安装 Ltib

请按照以下步骤在你的主机上安装 LTIB:

1. 用 root 登陆, 加载 ISO 镜像:  
`mount -o loop <target-bsp.iso> <mount point>`
2. 退出 root, 安装 LTIB:  
`<mount point>/install`

在安装过程中, 需要输入安装目标目录, 请确保用于安装 LTIB 的用户有安装目标目录的完全使用权限。

安装完 LTIB 之后, 目前暂不提供卸载工具或脚本。如果想要删除 LTIB, 那么请手工删除 /opt/freescale/pkgs、/opt/freescale/和<install\_path>/ltib 目录。

按照完后, 交叉编译器被安装到/opt/freescale/usr/local/<gcc version>/m68k-linux/bin 目录下。建议将此目录加入系统路径中。

### ● Ltib 使用

请用非 root 用户登陆, 转到 LTIB 安装目录运行“./ltib -c”

```
cd <install_path>/<ltib-modelo>
./ltib -c
```

注意: 第一次运行 ltib 会在主机上编译安装一些必要的软件包, 完成编译安装可能需要较长时间, 请耐心等待。这里 <ltib-modelo> 指 ltib 发布的名称, 对于当前来说, 表示 non-distributable-ltib-modelo-20100708, 在正式发布 ltib 包后, 名称会有改变。

在运行 ./ltib -c 之后将进入 platform/board 配置页面, 可以在配置页面下更改一些必要配置:

1. 选中 Build a boot loader 来生成 uboot  
集中器参考设计上默认的 bootmod[1:0]是 0b11 即从 SBF 启动, 且输入时钟为 50MHz, 所以在这里请选择 u-boot 的启动类型为 SPI boot-50M clocksource, 如图 1 所示:

```

--- LTIB settings
System features --->
--- Choose the target C library type
target C library type (glibc) --->
C library package (from toolchain only) --->
Toolchain component options --->
--- Toolchain selection.
Toolchain (gcc-4.4.54 eglibc-2.10.54 mk68k toolchain.) --->
(-march=isaac -mcpu=54418 -msoft-float) Enter any CFLAGS for gcc/g++
--- Bootloader
[*] build a boot loader
      u-boot target platform type
      Use the arrow keys to navigate this window or press the hotkey of
      the item you wish to select followed by the <SPACE BAR>. Press
      <?> for additional information about this option.
      ( ) AND boot-50M clocksource
      ( ) AND boot-25M clocksource
      ( ) RAM boot-50M clocksource
      ( ) RAM boot-25M clocksource
      (X) SPI boot-50M clocksource
      ( ) PI boot-25M clocksource
      <select> < Help >
---
Load an Alternate Configuration File
Save Configuration to an Alternate File

```

图 1 Uboot 编译选项

## 2. Target System Configuration→

这里可以配置内核启动后一些自动启动的项目，以及 Network setup→，可以根据实际需求配置一个或两个以太网以及 ip 地址，参考如图 2：

```

[*] enable interface 0
(eth0) interface
[*] get network parameters using dhcp
---
[*] enable interface 1
(eth1) interface
[ ] get network parameters using dhcp
(192.168.1.254) IP address
(255.255.255.0) netmask
(192.168.1.255) broadcast address
(192.168.1.1) gateway address
(192.168.1.1) netserver IP address
(udhcpd -b -i ) DHCP client startup

```

图 2 目标系统的网络配置

## 3. Package selection→

可以选择编译安装额外的软件包到 rootfs。

**注意：请务必选中[\*]Leave the sources after building。**以便编译完成之后可以对原内核打针对集中器板的补丁。

如果选中了 Configure the kernel，那么在退出 LTIB 配置页面后，将会进入 kernel menuconfig 界面，以配置 kernel 和 BSP。

首次运行 ltib -c 命令完成后，将集中器参考平台所提供的 kernel 补丁文件

kernel-2.6.29-modelo\_concentrator\_<release\_date>.patch 拷贝到

<install\_path>/<ltib-modelo>/rpm/BUILD 目录下，运行以下命令，更新为针对集中器板的 linux BSP：

```
patch -p0 < kernel-2.6.29-modelo_concentrator_<release_date>.patch
```

打完补丁后，将我们另外提供的为集中器 demo 配置的 linux-2.6.29-modelo.config.dev 文件拷贝到<install\_path>/<ltib-modelo>/config/platform/modelo/目录下。

接下来，准备应用程序，参考平台提供了一个简单的 DEMO 测试程序 ertu 供用户参考：

```
cd <install_path>/<ltib-modelo>/merge  
mkdir usr
```

请将 demo 程序 ertu 拷贝到<install\_path>/<ltib-modelo>/merge/usr/目录下

```
cd <install_path>/<ltib-modelo>/merge  
mkdir etc  
mkdir etc/rc.d
```

请将 rcS 文件拷贝到<install\_path>/<ltib-modelo>/merge/etc/rc.d/目录下

```
cd <install_path>/<ltib-modelo>/merge  
mkdir var  
mkdir var/www  
mkdir var/www/html
```

请将发布的包中 Web 目录下的所有文件拷贝到 html 目录下

以上步骤在再次运行./ltib 命令之后，可以将 ertu 和 rcS 文件生成到 rootfs 中。

编译安装完成后，将在<install\_path>/目录下生成以下文件和根文件系统目录：

- **rootfs/**: linux 根文件系统
- **rootfs/boot/ulmage**: 内核，由 u-boot 启动加载
- **rootfs/boot/u-boot.bin**: u-boot 二进制文件，直接用于烧写更新 u-boot
- **rootfs.jffs2**: jffs2 根文件系统镜像，直接用于烧写 NAND-Flash 上的 jffs2 文件系统

● LTIB 命令简要说明：

➤ **./ltib -h**

查看 ltib 命令帮助。

➤ **./ltib -c**

配置并编译安装内核、根文件系统以及选中的软件包。

➤ **./ltib [-m <mode>] [options...]**

ltib 命令基本格式，常用命令如下：

**./ltib -m prep -p <package>**: 准备指定 package，并不编译安装。

**./ltib -m scbuild -p <package>**: 编译指定 package。

**./ltib -m sdeploy -p <package>**: 安装指定 package 到 rootfs。

**./ltib -m config**: 进入配置页面，不进行编译。

**./ltib -m clean**: 清除生成的目标文件

**./ltib -m distclean**: 彻底删除所有生成文件，包括一些配置，请谨慎使用。

**./ltib -m patchmerge -p <package>**: 用于生成软件包补丁。首先更改指定 package 源代码目录（用户已更改过），然后执行./ltib -m prep -p <package>从原 LTIB 镜像中提出指定软件包，接着用 diff 命令生成补丁，最后清除原软件包（保留用户更改过的源代码）。需要注意的是，在执行此命令前，先用./ltib -m distclean -p <package>彻底清除所有生成文件，否则生成的目标文件也会被生成到补丁中去。

注：“-p <package>”表示指定 package 并只对指定 package 进行相应操作；以上 <package>可以是 kernel、u-boot，或者<install\_path>/non-distributable-ltib-modelo-20100708/dist/lfs-5.1/目录下的任一软件包 spec。

● 目标板和主机配置

➤ 目标板配置

电源: J6, DC-5V

网口(FEC1): J5

串口(ttyS0): J9, 波特率 115200, 数据 8 位, 无校验位, 1 停止位, 无流控。

## ➤ 主机配置

主机端的配置对于开发调试 BSP 相当关键，以下内容是通用配置，可能在不同的版本 linux 系统上会有些差别。

1. 关闭系统防火墙（需 root 权限），以确保 tftp 和 nfs 服务能正常工作：

console 下运行 **iptables -F** 或 **setup**(去配置页面关闭防火墙)

2. 安装 tftp 服务

3. 安装 nfs 服务

4. 建立 rootfs 目录链接：

```
mkdir /tftpboot
```

```
ln -s <install_path>/<ltib-modelo>/rootfs /tftpboot
```

注意：ln 命令需要用完整路径

5. 拷贝 ulmage、u-boot.bin 和 rootfs.jffs2 到/tftpboot 目录

```
cp <install_path>/<ltib-modelo>/rootfs/boot/ulmage /tftpboot
```

```
cp <install_path>/<ltib-modelo>/rootfs/boot/u-boot.bin /tftpboot
```

```
cp <install_path>/non-distributable-ltib-modelo-20100708/rootfs.jffs2 /tftpboot
```

6. 编辑/etc/exports 文件（需 root 权限），加入以下内容（配置 nfs 服务根目录）  
/tftpboot/rootfs \*(rw,no\_root\_squash,async)

注意：请用 root 编辑/etc/exports，输入 vi /etc/exports

7. 编辑/etc/xinetd.d/tftp，配置 tftp 服务（需 root 权限），如下所示：

```
{  
  disable = no  
  socket_type = dgram  
  protocol = udp  
  wait = yes  
  user = root  
  server = /usr/sbin/in.tftpd  
  server_args = /tftpboot  
}
```

8. 重启 nfs 和 tftp 服务（需 root 权限）

```
/etc/init.d/xinetd restart
```

```
service nfs restart
```

9. 配置主机 ip 为 192.168.1.1，配置完成后运行以下命令重启网络服务（需 root 权限）：

**service network restart**

10. 用交叉网线连接主机和目标板

11. 用交叉串口线连接主机和目标板

12. 在主机端开启串口终端 minicom，设置如下：115200 8-N-1

13. 目标板接入 DC-5V 电源

## ● 烧写和更新 u-boot

目标（集中器）板可支持从 SBF 和 NAND-Flash 启动。

- ### ➤ 首次烧写从 SBF 启动的 u-boot.bin

1. 首先将 u-boot.bin 从 linux 系统中 copy 到 windows 系统下，并在 u-boot.bin 头上加入 7 个 16 进制字节：03 ff 00 61 fc 09 65

相关内容可参考 MCF54418RM 第 11 章 Serial Boot Facility 和 10.4.1.3 Reset Configuration (BOOTMOD[1:0]=11)

u-boot.sbf.bin 的头可以在 LTIB 镜像中找到

M54418Tower\_Linux\_BSP\_20100618\_ltib.iso\images，可用以下命令组合头和

u-boot.bin：

```
cat sbfhdr.54418 > u-boot.sbf.bin
```

```
cat u-boot.bin >> u-boot.sbf.bin
```

2. 解压 SBF 烧写工具：

M54418Tower\_Linux\_BSP\_20100618\_itib.iso\help\software\CFFlasher\CFFlasher-sbf-mram.zip

3. 在 Windows 环境下运行 CFFlasher.exe, 如图 3 所示。使用 P&E BDM 连接到目标板上的 BDM 接头(2x13 插针), 并通过 USB 线把 P&E BDM 和电脑连接起来。



图 3 CFFlasher 启动

4. Target Config, 请按图 4 配置:

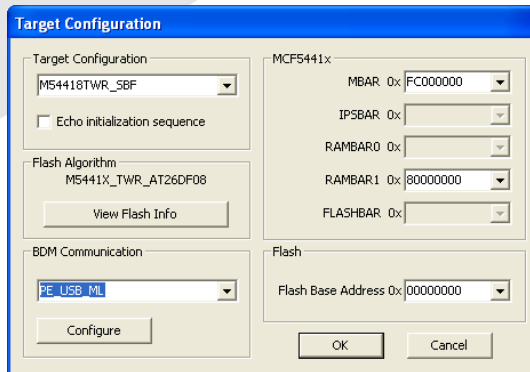


图 4 CFFlasher 烧写目标配置

5. Erase, 如图 5:

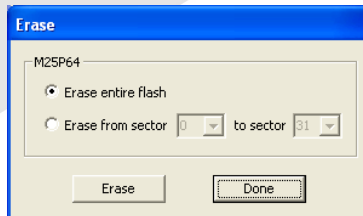


图 5 擦除

6. Program, 如图 6:

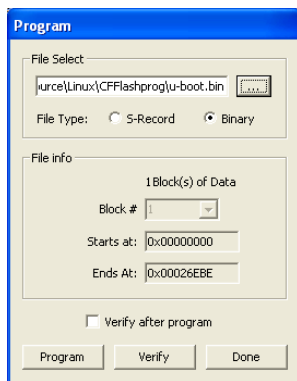


图 6 CFFlasher 编程

起始地址为 0

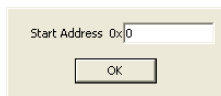


图 7 编程起始地址配置

注：结束地址依赖于 u-boot.sbf.bin 文件的大小。

- 用 u-boot 更新从 SBF 启动的 u-boot.bin（目标板已可以从 SBF 或 NAND-flash 启动）
- 1. 用交叉串口线连接目标板和主机
- 2. 用交叉网线连接目标板和主机
- 3. 接入 DC-5V 电源
- 4. 等待 u-boot 启动完毕（如果设置了 bootcmd，请在 timeout 之前停止内核自启动）
- 5. 在串口控制台下请依次输入以下命令：

```
=>sf probe 0:1 1000000 3
=>sf erase 0 1
00000
=>tftp 40010007 u-boot.bin
=>mw.l 40010000 03ff0061
=>mw.l 40010004 fc096500
=>sf write 40010000 0 30000
=>sf read 40050000 0 30000
=>cmp.b 40010000 40050000 30000
```

注意：如果所烧写的 Kernel image 已经添加了 SBF 启动用的 7 个字节，则运行以下命令：

```
=>sf probe 0:1 1000000 3
=>sf erase 0 100000
=>tftp 40010000 u-boot.sbf.bin
=>sf write 40010000 0 30000
=>sf read 40050000 0 30000
=>cmp.b 40010000 40050000 30000
```

- 首次烧写从 NAND-Flash 启动的 u-boot.bin
- 首先需要把核心板配置成 Nand Flash 启动方式（具体见硬件手册）。因为目标（集中器）板与飞思卡尔发布的 LTIB 的目标板略有差异，所以需要先给 u-boot 打个补丁：
1. 转到 LTIB 安装目录（<install\_path>/<ltib-modelo>/）下，准备原镜像 u-boot：  
./ltib -m prep -p u-boot-2009.08-modelo.spec
  2. cd rpm/BUILD/
  3. 拷贝 u-boot 补丁文件至当前目录(<install\_path>/<ltib-modelo>/rpm/BUILD/)
  4. patch -p0 <u-boot-2009.08-concentrator\_nand\_boot.patch

5. cd u-boot-2009.08
6. make M54418TWR\_nand\_rmii\_config
7. make ARCH=m68k  
CROSS\_COMPILE=/opt/freescale/usr/local/gcc-4.4.54-eglibc-2.10.54/m68k-linux/bin/m68k-linux-gnu- -j4
8. 在 Windows 环境下解压 NAND-Flash 烧写工具：  
M54418Tower\_Linux\_BSP\_20100618\_Itib.iso\help\software\CFFlasher\CFFlasher-nand.zip
9. 将编译得到的 u-boot.bin 拷贝至 NAND-Flash 工具目录下
10. 将 USB Coldfire Multilink(BDM)接到目标板 BDM 接头（2x13 插针），并用 USB 线连接到 PC，接入 DC-5V 电源
11. 在 windows 系统下运行 cmd
12. 到烧写工具的目录下按照烧写 NAND-Flash 工具的 readme.txt，依次执行以下命令：

```
cf nand erase m54418twr_nand 0 40000
cf nand write m54418twr_nand 0 40000 1 u-boot.bin
```

- 用 u-boot 更新从 NAND-Flash 启动的 u-boot.bin（目标板已可以从 SBF 或 NAND-Flash 启动）

1. 拷贝编译所得 u-boot.bin 到/tftpboot（参考上节内容 1-7）
2. 启动目标板
3. 依次在 u-boot 下输入以下命令：

```
=>tftp 41000000 u-boot.bin
=>nand erase 0 40000
=>nb_update 41000000 ${filesize}
```

其中\${filesize}是第一条指令 tftp 传输的实际 u-boot.bin 文件长度

注意：如果在 erase 时，如果出现  
nand\_read\_bbt: Bad block at 0x00000000  
nand\_read\_bbt: Bad block at 0x00020000

...

Skipping bad block at 0x00000000  
Skipping bad block at 0x00020000

表示第 0,1 个 Block 可能被 cf nand 工具误标为坏块。此时可以通过全芯片擦除来重新标记坏块。运行：

```
=>nand scrub
```

- u-boot 配置

请在 u-boot 下依次输入以下命令来配置 ip 地址以及启动脚本：

```
=>set ipaddr 192.168.1.254
=>set serverip 192.168.1.1
```

```
=>set nand_boot 'setenv bootargs root=/dev/mtdblock2 rw rootfstype=jffs2
mtdparts=NAND:1M(boot)ro,7M(kernel),-(jffs2) console=ttyS0,115200;nand read 0x42000000 0x400000
400000;bootm 42000000'
```

如果需要采用 nfs 加载启动内核并加载 rootfs，请输入以下命令：

```
=>set nfs_boot 'set ethact FEC1;set bootargs root=/dev/nfs rw nfsroot=192.168.1.1:/tftpboot/rootfs
ip=192.168.1.254:192.168.1.1::freescale:eth1:off mtdparts=NAND:1M(boot)ro,7M(kernel),-(jffs2)
console=ttyS0,115200;tftp 44000000 ulmage;bootm 44000000'
```

```
=>save
```

- 烧写 linux 内核和 jffs2 根文件系统

首先将编译所得 `ulmage` 和 `rootfs.jffs2` 拷贝至 `/tftpboot/` 目录下，连上网线和串口线，开启串口中断，将目标板上电启动到 `u-boot` 界面下。

依次运行以下命令烧写 linux 内核：

```
=>tftp 41000000 ulmage
=>nand erase 400000 400000
=>nand write 41000000 400000 400000
```

依次运行以下命令烧写 `rootfs.jffs2`：

```
=> tftp 41000000 rootfs.jffs2
=> nand erase 800000
=> nand write.jffs2 41000000 800000 ${filesize}
```

完成烧写后，输入 `run nfs_boot` 或 `run nand_boot` 即可启动 linux。

- 开发用户自己的驱动

1. 编写自己的驱动和 Makefile，可参考 2.2.11 节的 Makefile

2. 编译：

转到 Makefile 所在目录，输入 `“make ARCH=m68k CROSS_COMPILE=/opt/freescale/usr/local/gcc-4.4.54-eglibc-2.10.54/m68k-linux/bin/m68k-linux-gnu-”` 即可生成驱动.ko 文件

3. 拷贝.ko 文件到 nfs rootfs 下或

`<install_path>/non-distributable-ltlib-modelo-20100708/merge` 目录，通过 `./ltib` 生成到 `rootfs.jffs2`

4. 启动 linux 后用 `insmod`、`lsmod`、`rmmmod` 命令测试

- 编译生成到内核

上节内容简单介绍了如何开发并编译生成自己的驱动.ko 文件，本节内容将简单介绍如何把自己的驱动编译到 linux 内核中，即 linux 启动后自动加载驱动而无需 `insmod` 命令。以 `<install_path>/<ltib-modelo>/rpm/BUILD/linux/drivers/Concentrator/` 中的驱动为例：

1. `mkdir linux/drivers/Concentrator`

2. 拷贝驱动文件到上述目录

3. 在 `linux/drivers/Kconfig` 文件中添加：

```
source "drivers/Concentrator/Kconfig"
```

4. 在 `linux/drivers/Makefile` 文件中添加：

```
obj-y += Concentrator/
```

5. 创建 `linux/drivers/Concentrator/Kconfig` 文件，并添加以下内容：

```
#
# Concentrator Demo Drivers
#

menu "Concentrator Demo Drivers"

config ST7529_LCDC
bool "ST7529 LCDC support"
default y
help
  Say yes will support ST7529 LCDC

config CONCENTRATOR_EXT_GPIO
bool "Extern GPIO Support for LEDs"
default y
help
  Say yes will include ext_gpio to support LEDs on board.

config CONCENTRATOR_KEYPAD
bool "Keypad Support"
default y
```

*help*  
Say yes will support keypads on board.

*endmenu*

6. 创建 linux/drivers/Concentrator/Makefile 文件，并添加以下内容：  
`obj-$(CONFIG_ST7529_LCDC) += st7529lcd.o hzk12x12.o winFreeSansSerif11x13.o`  
`obj-$(CONFIG_CONCENTRATOR_EXT_GPIO) += Concentrator_extGPIO.o`  
`obj-$(CONFIG_CONCENTRATOR_KEYPAD) += Concentrator_Keypad.o`

这样，在 *make menuconfig* 时，就会在 kernel 配置页面 Devices Driver→下出现“Concentrator Demo Drivers→”。

**Kconfig:** 此文件在运行 *menuconfig* 时被读取，根据用户选择定义编译内核时所需必要宏定义。

**Makefile:** 根据相同目录下的 Kconfig 文件定义的宏，添加链接时所需的.o 文件。

更多内容，可参考各个模块目录下的 Kconfig 和 Makefile 文件。

- 用户如何添加自己的应用程序

这里以 ERTU\_APP 为例：

1. 在 Linux 环境下编写应用和 Makefile
2. 将应用程序源代码目录拷贝到<install\_path>/<ltib-modelo>/rpm/BUILD 目录下
3. 在应用程序源代码目录下运行“*make ARCH=m68k CROSS\_COMPILE=/opt/freescale/usr/local/gcc-4.4.54-eglibc-2.10.54/m68k-linux/bin/m68k-linux-gnu-*”，即可生成目标文件 src/ertu
4. 直接拷贝 ertu 文件到 rootfs 目录或<install\_path>/<ltib-modelo>/merge/usr/目录下并通过运行 ./ltib 将其生成到 rootfs 中。
5. 通过 nfs 启动或者烧写 jffs2 文件系统到目标板再启动后即可运行用户程序。

注：如果代码未拷贝到上述目录下，请自行更改 src/Makefile 中 CFLAGS 变量关于头文件目录设置：

```
CFLAGS = -I../linux/include -g -Os -pipe -Wall
```

## 2.2 驱动说明

### 2.2.1 NandFlash 驱动

参考平台上使用的是 Micron 29F2G16AAD，256MB，16bit NAND-Flash。要使能 NFC 驱动，需要在内核配置中使能以下选项。

```
--Linux Kernel Configuration--
Device Drivers →
<*> Memory Technology Device (MTD) support →
<*> NAND Device Support →
<*> Support for NAND on Freescale Coldfire NFC
```

另外，为了支持在 NAND-Flash 上分区以存放 u-boot、kernel image 和 JFFS2 文件系统，需要使能以下选项。

```
--- Memory Technology Device (MTD) support
<*> MTD partitioning support
[*] Command line partition table parsing
<*> Direct char device access to MTD devices
<*> Caching block device access to MTD devices
```

参考平台默认采用的是从 SBF 启动，即 u-boot 存放于 SBF 中，而 kernel image(ulmage)和

rootfs(jffs2)则存放于 NAND-Flash 中，其中 ulmage 存放于 mtdblock1 而 jffs2 rootfs 则存放于 mtdblock2。(如果选择了从 NAND-Flash 启动，则 u-boot 是存放于 mtdblock0 中。)

系统启动后，可以用“cat /proc/partitions”查看加载 MTD 的状况：

```
[root@M54418TWR block]# cat /proc/partitions
major minor #blocks name
31 0 1024 mtdblock0
31 1 7168 mtdblock1
31 2 253952 mtdblock2
31 3 1024 mtdblock3
```

注意：其中 mtdblock3 是 SBF，从启动信息中也可看到：

```
FSL NFC MTD nand Driver 0.5
NAND device: Manufacturer ID: 0x2c, Chip ID: 0xca (Micron NAND 256MiB 3,3V 16-bit)
3 cmdlinepart partitions found on MTD device NAND
Creating 3 MTD partitions on "NAND":
0x000000000000-0x000000100000 : "u-boot"
0x000000100000-0x000000800000 : "kernel"
0x000000800000-0x000010000000 : "jffs2"
m25p80 spi.1: at26df081a (1024 Kbytes)
Creating 1 MTD partitions on "Atmel at26df081a SPI Flash chip":
0x000000000000-0x000000100000 : "at26df081a"
```

注意：如果调试过程中以 NFS 方式加载 rootfs，请勿 mount 除 mtdblock2 以外的 mtd 设备，以免误擦写只读区域。

## 2.2.2 DSPI 驱动

DSPI 驱动在内核中选择配置。在参考平台中，SPI0 上挂载了 SPI Flash，用户程序可以通过 DSPI 驱动对此设备进行操作。要支持 DSPI 设备，在编译内核时，需要选择对应选项。

```
--Linux Kernel Configuration--
Device Drivers->SPI support
<*>Coldfire DSPI
```

并由于参考设计中使用 DSPI0 端口连接串行 flash，所以这里选择 DSPI0 作为 DSPI 控制器。如图所是，原理图中使用了 DSPI0\_PCS1 作为串行 flash 的片选信号，因此把 0x1 作为配置数据。

```
(0x1) Chip select for serial flash on DSPI0
```

这里还选择了

```
<*> User mode SPI device driver support
```

该选项是用于支持用户自己写的 api 代码对/dev 目录下的 spi 设备进行打开关闭和读写操作的功能，如果没有该需求，可以不选。

如果要使能 DSPI 的 DMA 功能，可以把 eDMA 功能选上。

```
[*]Coldfire DSPI master driver uses eDMA
```

## 2.2.3 以太网驱动

参考平台上以太网 eth0 直接连接到集中器的本地通信接口上，eth1 引到 RJ45 上，因此可以直接通过网线来测试以太网 eth1，而以太网 eth0 则需要本地通信模块来连接。BSP 内部已经集成了以太网驱动代码。客户可以很方便的采用标准的 Linux 以太网设备 eth0/1 来使用以太网模块。

采用 MCF5441x 的 eth1 口，用户需要通过 Itib 做以下配置，这里通过 DHCP 动态地获取板子的 IP 地址。

- DHCP 的使用

```
--- Target System Configuration
Options -> Network Setup ->
[*] Enable interface 1
```

```
(eth1) interface
[*] get network parameters using DHCP
(udhcpc -b -i) DHCP client startup
```

将板子连到路由器上后上电，在 Linux 启动结尾处可以看到如下输出，表示板子获得了 IP 地址为 192.168.1.100。

```
Setting up networking on eth1:
udhcpc (v1.11.2) started
attached phy 1 to driver NatSemi DP83849
Sending discover...
PHY: 0:01 - Link is Up - 100/Full
Sending discover...
Sending select for 192.168.1.100..
Lease of 192.168.1.100 obtained, lease time 7200
```

**注意：**通过 DHCP 获取 IP 地址是和板子的 MAC 地址相关的，两个具有相同 MAC 地址的板子会获得相同的 IP 地址。用户可以在 uboot 中修改板子的

MAC 地址，命令如下：

```
set eth1addr 00:e0:0c:bc:e5:62 //修改 MAC 地址
save //保存环境变量
```

#### ● netperf

Netperf 主要用于对网络性能进行测量，它是基于 client 和 server 的结构，其软件包生成两个工具，netserver（作为 server）和 netperf（作为 client）。在一个由参考平台组成的子网内，可以将其中的一个作为 server，在该板子上运行以下命令，启动 netserver 来侦听 12865 端口的通信。

```
[root@M54418TWR /]# netserver
Starting netserver at port 12865
Starting netserver at hostname 0.0.0.0 port 12865 and family AF_UNSPEC
```

然后在作为 client 的其他参考平台上运行以下命令来检测当前网络的性能，吞吐量等数据。

关于 netperf 的一些参数和输出数据的含义，请参照

<http://www.ibm.com/developerworks/cn/linux/l-netperf/>。

```
[root@M54418TWR /]# netperf -H 192.168.1.101 -l 60
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 192.168.1.101 (192.168.1.101) port 0 AF_INET
Recv  Send  Send
Socket Socket  Message  Elapsed
Size  Size  Size  Time  Throughput
bytes bytes bytes secs.  10^6bits/sec
 87380 16384 16384   60.01    67.69
```

#### ● boa webserver

参考平台系统启动后，就运行了自带的 boa webserver，用户只需通过一个浏览器就可以访问板子运行的 webserver 所带的网页，在地址栏中键入板子的 IP 地址即可。需要注意的是，应该关掉浏览器中使用的代理服务器，否则无法访问到目标板的网页。缺省的网页位于目标板的 /var/www/html 目录下，用户可以替换为自己的网页。

## 2.2.4 I2C 驱动

MCF5441x 上拥有 I2C 控制器，要使用这些 I2C 接口，需要在内核的驱动中使能对应的选项。

在内核配置菜单中：

```
--Linux Kernel Configuration--
Device Drivers →
<*> I2C support →
<*> I2C device interface
```

以及

```
I2C Hardware Bus support →
```

<\*>MCF ColdFire I2C Interface

这样在设备目录就可以加载 I2C 总线上的设备了。

## 2.2.5 RTC 驱动

MCF5441x 内部集成了一个 RTC 模块，同时参考设计上还配了一颗 I2C-RTC 芯片：Epson RX-8025T 实现外部的 RTC 芯片功能。LinuxBSP 提供了两个 RTC 的驱动，用户可以按照自己的需求自行选择使用其中一个或两个。所有 RTC 相关的配置都在内核驱动的菜单目录下：

--Linux Kernel Configuration--

Device Drivers → Real Time Clock →

要用 RTC 来配置系统时间，要选择

[\*] Set system time from RTC on startup and resume

(rtc0) RTC used to set the system time

[\*] /sys/class/rtc/rtcN (sysfs)

[\*] /proc/driver/rtc (procfs for rtc0)

[\*] /dev/rtcN (character devices)

要使用内部 RTC，需要选择该菜单下的片上 RTC 选项

<\*>Freescale Coldfire M5441X platform Real Time Clock

要使用外部 RTC，除了需使能上节内容提到的 I2C 驱动配置外，要使用该芯片还需使能该菜单下的 Rx-8025 选项。

<\*>RX-8025 IIC Real Time Clock

在使能 RX-8025 IIC Real Time Clock(RTC\_RX8025)之后，如果同时使能了该页面最后的 Freescale Coldfire M5441X platform Real Time Clock(RTC\_M5441X)，那么在 kernel 启动后，RTC\_RX8025 将对应/dev/rtc1，而 MCF5441x 内部 RTC 模块将对应/dev/rtc0。建议不使用 MCF5441x 内部 RTC，这样，RTC\_RX8025 则对应/dev/rtc0。

用户如要更换其他类型 RTC 芯片作为高精度实时时钟，则需自行更改 devices.c(kernel 目录 \arch\m68k\coldfire\m5441x\ )中关于 RTC 芯片的初始化，这里以参考设计为例，介绍如何添加 RX-8025 作为高精度实时时钟：

```
static struct i2c_board_info rtc_rx8025_info = {
    I2C_BOARD_INFO("rx8025", 0x32),
    .irq = 71,
};
```

这里定义了 I2C-RTC 芯片的信息，即 name、I2C address 和使用到的外部中断号 (IRQ7)。然后需要在初始化 I2C 之后注册该 I2C-RTC 芯片以通知 kernel 将会有有一个 I2C 设备挂载到 I2C 总线上：

```
static void mcf5441x_init_i2c(void)
{
    MCF_PM_PPMCR1 = 4;
    platform_device_register(&coldfire_i2c_device);
    i2c_register_board_info(coldfire_i2c_device.id, &rtc_rx8025_info, 1);
}
```

### ● I2C-RTC 使用

RX-8025 驱动：linux 目录\driver\rtc\rtc-rx8025.c

API 介绍：

本驱动仅支持 1 分钟或 1 秒钟中断（不支持 RTC\_PIE\_ON），闹铃最小间隔为分钟。

ioctl 支持的 command 有：

RTC\_RD\_TIME, 读取 RTC 时间

RTC\_SET\_TIME, 设置 RTC 时间

RTC\_ALM\_READ, 读取闹铃时间  
 RTC\_ALM\_SET, 设置闹铃时间  
 RTC\_IRQP\_READ, 读取 IRQ 频率  
 RTC\_IRQP\_SET, 设置 IRQ 频率  
 RTC\_UIE\_OFF, 关闭 1s 或 1min 中断  
 RTC\_UIE\_ON, 开启 1s 或 1min 中断  
 RTC\_AIE\_ON, 开启闹铃  
 RTC\_AIE\_OFF, 关闭闹铃

其中 RTC\_IRQP\_READ 命令, 返回读取值为 0 或 1; RTC\_IRQP\_SET, 参数值只能是 0 或 1。0 代表 1min 中断间隔, 1 代表 1s 中断间隔。

如何使用可参考 rtc-test:

在 Itib 安装目录下, 输入“./Itib -p rtc-test -m prep”, 得到 rpm/BUILD/rtc-test/rtc-test.c。

#### ● I2C-RTC 测试

由于 RX-8025 驱动仅支持 1 分钟或 1 秒钟中断输出 (默认 1 分钟中断), 所以需要更改 rtc 测试程序 (请注意红色修改部分, 并注意是否使能了 MCF5441x 内部 RTC, 如果是, 请更改 default\_rtc[]="/dev/rtc1"):

```
#include <stdio.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
/* This expects the new RTC class driver framework, working with
 * clocks that will often not be clones of what the PC-AT had.
 * Use the command line to specify another RTC if you need one.
 */
static const char default_rtc[] = "/dev/rtc0";
int main(int argc, char **argv)
{
    int i, fd, retval, irqcount = 0;
    unsigned long tmp, data;
    struct rtc_time rtc_tm;
    const char *rtc = default_rtc;

    switch (argc) {
    case 2:
        rtc = argv[1];
        /* FALLTHROUGH */
    case 1:
        break;
    default:
        fprintf(stderr, "usage: rtctest [rtcdev]\n");
        return 1;
    }
    fd = open(rtc, O_RDONLY);
    if (fd == -1) {
        perror(rtc);
        exit(errno);
    }
    fprintf(stderr, "\n\t\t\tRTC Driver Test Example.\n\n");
    ioctl(fd, RTC_IRQP_SET, 1); //set to 1s interval, default is 1min interval
    /* Turn on update interrupts (one per second) */
    retval = ioctl(fd, RTC_UIE_ON, 0);
    if (retval == -1) {
        if (errno == ENOTTY) {
            fprintf(stderr,
                "\n...Update IRQs not supported.\n");
            goto test_READ;
        }
        perror("RTC_UIE_ON ioctl");
        exit(errno);
    }
}
```

```

fprintf(stderr, "Counting 5 update (1/sec) interrupts from reading %s:", rtc);
fflush(stderr);
for (i=1; i<6; i++) {
    /* This read will block */
    retval = read(fd, &data, sizeof(unsigned long));
    if (retval == -1) {
        perror("read");
        exit(errno);
    }
    fprintf(stderr, "%d", i);
    fflush(stderr);
    irqcount++;
}

fprintf(stderr, "\nAgain, from using select(2) on /dev/rtc:");
fflush(stderr);
for (i=1; i<6; i++) {
    struct timeval tv = {5, 0}; /* 5 second timeout on select */
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(fd, &readfds);
    /* The select will wait until an RTC interrupt happens. */
    retval = select(fd+1, &readfds, NULL, NULL, &tv);
    if (retval == -1) {
        perror("select");
        exit(errno);
    }
    /* This read won't block unlike the select-less case above. */
    retval = read(fd, &data, sizeof(unsigned long));
    if (retval == -1) {
        perror("read");
        exit(errno);
    }
    fprintf(stderr, "%d", i);
    fflush(stderr);
    irqcount++;
}
/* Turn off update interrupts */
retval = ioctl(fd, RTC_UIE_OFF, 0);
ioctl(fd, RTC_IRQP_SET, 0); //set to default value, 1min interval
if (retval == -1) {
    perror("RTC_UIE_OFF ioctl");
    exit(errno);
}
test_READ:
/* Read the RTC time/date */
retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);
if (retval == -1) {
    perror("RTC_RD_TIME ioctl");
    exit(errno);
}
fprintf(stderr, "\n\nCurrent RTC date/time is %d-%d-%d, %02d:%02d:%02d.\n",
    rtc_tm.tm_mday, rtc_tm.tm_mon + 1, rtc_tm.tm_year + 1900,
    rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);
/* Set the alarm to next min in the future, and check for rollover */
rtc_tm.tm_sec = 0;
rtc_tm.tm_min += 1;
if (rtc_tm.tm_sec >= 60) {
    rtc_tm.tm_sec %= 60;
    rtc_tm.tm_min++;
}
if (rtc_tm.tm_min == 60) {
    rtc_tm.tm_min = 0;
    rtc_tm.tm_hour++;
}
if (rtc_tm.tm_hour == 24)
    rtc_tm.tm_hour = 0;
retval = ioctl(fd, RTC_ALM_SET, &rtc_tm);
if (retval == -1) {
    if (errno == ENOTTY) {
        fprintf(stderr,
            "\n...Alarm IRQs not supported.\n");
        goto test_PIE;
    }
}
perror("RTC_ALM_SET ioctl");
exit(errno);
}

```

```

/* Read the current alarm settings */
retval = ioctl(fd, RTC_ALM_READ, &rtc_tm);
if (retval == -1) {
    perror("RTC_ALM_READ ioctl");
    exit(errno);
}
fprintf(stderr, "Alarm time now set to %02d:%02d:%02d.\n", rtc_tm.tm_hour, rtc_tm.tm_min,
rtc_tm.tm_sec);
/* Enable alarm interrupts */
retval = ioctl(fd, RTC_AIE_ON, 0);
if (retval == -1) {
    perror("RTC_AIE_ON ioctl");
    exit(errno);
}
fprintf(stderr, "Waiting less than 1min for alarm...");
fflush(stderr);
/* This blocks until the alarm ring causes an interrupt */
retval = read(fd, &data, sizeof(unsigned long));
if (retval == -1) {
    perror("read");
    exit(errno);
}
irqcount++;
fprintf(stderr, " okay. Alarm rang.\n");
/* Disable alarm interrupts */
retval = ioctl(fd, RTC_AIE_OFF, 0);
if (retval == -1) {
    perror("RTC_AIE_OFF ioctl");
    exit(errno);
}
test_PIE:
/* Read periodic IRQ rate */
retval = ioctl(fd, RTC_IRQP_READ, &tmp);
if (retval == -1) {
    /* not all RTCs support periodic IRQs */
    if (errno == ENOTTY) {
        fprintf(stderr, "\nNo periodic IRQ support\n");
        goto done;
    }
    perror("RTC_IRQP_READ ioctl");
    exit(errno);
}
fprintf(stderr, "\nPeriodic IRQ rate is %ldHz.\n", tmp);
fprintf(stderr, "Counting 20 interrupts at:");
fflush(stderr);
/* The frequencies 128Hz, 256Hz, ... 8192Hz are only allowed for root. */
for (tmp=2; tmp<=64; tmp*=2) {
    retval = ioctl(fd, RTC_IRQP_SET, tmp);
    if (retval == -1) {
        /* not all RTCs can change their periodic IRQ rate */
        if (errno == ENOTTY) {
            fprintf(stderr,
                "\n...Periodic IRQ rate is fixed\n");
            goto done;
        }
        perror("RTC_IRQP_SET ioctl");
        exit(errno);
    }
    fprintf(stderr, "\n%ldHz:\t", tmp);
    fflush(stderr);
    /* Enable periodic interrupts */
    retval = ioctl(fd, RTC_PIE_ON, 0);
    if (retval == -1) {
        perror("RTC_PIE_ON ioctl");
        exit(errno);
    }
}
for (i=1; i<21; i++) {
    /* This blocks */
    retval = read(fd, &data, sizeof(unsigned long));
    if (retval == -1) {
        perror("read");
        exit(errno);
    }
    fprintf(stderr, "%d", i);
    fflush(stderr);
    irqcount++;
}

```

```

    }
    /* Disable periodic interrupts */
    retval = ioctl(fd, RTC_PIE_OFF, 0);
    if (retval == -1) {
        perror("RTC_PIE_OFF ioctl");
        exit(errno);
    }
}
done:
    fprintf(stderr, "\n\n\t\t\t *** Test complete ***\n");
    close(fd);
    return 0;
}

```

修改完之后，仍然在 Itib 安装目录下输入“./Itib -m scbuild -p rtc-test”编译 rtc-test 程序，然后输入“./Itib -m scdeploy -p rtc-test”将 rtc-test 可执行程序装载到 rootfs 中。接下去就可以运行测试了（通过 NFS 加载 rootfs 或者重新将 rootfs.jffs2 烧写到 NAND-Flash 中）：

# rtc-test

*RTC Driver Test Example.*

*Counting 5 update (1/sec) interrupts from reading /dev/rtc0: 1 2 3 4 5  
Again, from using select(2) on /dev/rtc: 1 2 3 4 5*

*Current RTC date/time is 18-6-2010, 13:13:58.  
Alarm time now set to 13:14:00.  
Waiting less than 1min for alarm... okay. Alarm rang.*

*Periodic IRQ rate is 0Hz.  
Counting 20 interrupts at:RTC\_IRQP\_SET ioctl: Invalid argument*

## 2.2.5 UART 驱动

- 驱动说明

MCF5441x 共有 10 个串口，由于引脚复用等原因，在集中器参考平台中，使用有 8 个串口。其分配如表 1

串口	功能
U0 (不带 RTS/CTS)	Linux 串口控制台
U1 (带 RTS/CTS)	远程通信模块 Modem
U2 (不带 RTS/CTS)	RS485
U4	预留，未使用
U6	红外接口
U7	RS485
U8	本地通信模块 DCE 信号
U9	RS485

表 1 串口分配

在 LinuxBSP 环境下，可以通过配置内核的串口来使能。

--Linux Kernel Configuration--  
Device Drivers →Character devices →

选中以下几个，用来使能控制台输出到串口上

- [\*] Virtual terminal
- [\*] Enable character translations in console
- [\*] Support for console on virtual terminal
- [\*] Unix98 PTY support

进入 Serial drivers →如图 8

```

< > 8250/16550 and compatible serial support
*** Non-8250 serial port support ***
[*] ColdFire IRDA support
[4] Default IRDA Line
[ ] ColdFire serial EDMA support
[*] Coldfire serial support
(115200) Default baudrate for Coldfire serial ports
[*] Coldfire serial console support
[*] Coldfire UART0 device support
[*] Coldfire UART1 device support
[*] Coldfire UART2 device support
[ ] Coldfire UART3 device support
[*] Coldfire UART4 device support
[ ] Coldfire UART5 device support
[*] Coldfire UART6 device support
[*] Coldfire UART7 device support
[*] Coldfire UART8 device support
[*] Coldfire UART9 device support
  
```

图 8 串口配置

系统中配置缺省的串口波特率为 115200，并且使能了除 UART3,5 之外的所有串口。由于跳过了 UART3 和 5，因此串口 UART6 被系统自动认为 ttys4 设备，后面的依次类推。ColdFire IRDA support 是用来使能红外的串口，由于硬件上使用 UART6 来做红外设备，因此在 Default IRDA Line 上要选择 4，表示对应 ttys4，用来给红外驱动传递参数。如果用户有其他的串口配置改变，这里也需要做相应修改。

- 使用串口驱动

打开串口

在 Linux 下串口文件是位于 /dev 下的。串口一为 /dev/ttyS0，串口二为 /dev/ttyS1。打开串口是通过使用标准的文件打开函数操作：

```

int fd;
/*以读写方式打开串口*/
fd = open( "/dev/ttyS0", O_RDWR);
if (-1 == fd){
/* 不能打开串口一*/
perror(" 提示错误! ");
}
  
```

设置串口

最基本的设置串口包括波特率设置，校验位和停止位设置。串口的设置主要是设置 struct termios 结构体的各成员值。

```

struct termio
{
  unsigned short c_iflag; /* 输入模式标志 */
  unsigned short c_oflag; /* 输出模式标志 */
  unsigned short c_cflag; /* 控制模式标志 */
  unsigned short c_lflag; /* local mode flags */
  unsigned char c_line; /* line discipline */
  unsigned char c_cc[NCC]; /* control characters */
};
  
```

读写串口

设置好串口之后，读写串口就很容易了，把串口当作文件读写就是。

发送数据

```

char buffer[1024];
int Length;
int nByte;
nByte = write(fd, buffer, Length)
  
```

读取数据

使用文件操作 read 函数来读取串口，如果设置为原始模式(Raw Mode)传输数据，那么 read 函数返回的字符数是实际串口收到的字符数（即包括回车/换行符）。可以使用操作文件的函

数来实现异步读取，如 `fcntl`，或者 `select` 等操作。

```
char buff[1024];
int Len;
int readByte;
readByte = read(fd,buff,Len);
```

应用样例

下面是一个简单的读取串口数据的例子，使用了上面定义的一些函数和头文件

```

/*****
代码说明：使用串口二测试的，发送的数据是字符，
但是没有发送字符串结束符号，所以接收到后，后面加上了字符串结束符号。
*****/
#define FALSE -1
#define TRUE 0
/*****/
int OpenDev(char *Dev)
{
    int fd = open( Dev, O_RDWR | O_NOCTTY | O_NDELAY);
    if (-1 == fd)
    {
        perror("Can't Open Serial Port");
        return -1;
    }
    else
        return fd;
}

int main(int argc, char **argv)
{
    int fd;
    int nread;
    char buff[512];
    char *dev = "/dev/ttyS1"; //串口二
    fd = OpenDev(dev);
    set_speed(fd,19200);
    if (set_Parity(fd,8,1,'N') == FALSE)
    {
        printf("Set Parity Error\n");exit (0);
    }
    while (1) //循环读取数据
    {
        while((nread = read(fd, buff, 512))>0)
        {
            printf("\nLen %d\n",nread);
            buff[nread+1] = '\0';
            printf( "\n%s", buff);
        }
    }
    close(fd);
    exit (0);
}

```

## 2.2.6 USB 驱动

MCF5441x 系列拥有两个 USB 模块，一个 USB Host 和一个 USB Host/Device/On-The-Go 模块。参考平台上将 USB Host/Device/OTG 作为远程通信模块的接口，而 USB Host 模块作为备用。目前 BSP 提供了 USB Host/Device/OTG 的驱动，在下一版中将提供 USB Host 模块的驱动。参考平台的 USB 驱动支持 HOST 模式和大规模存储设备。以下是通过 kernel menuconfig 配置 USB 驱动所需的一些设置，注意这里选择的是片上的 FS/LS USB 收发器，而所有的相关模块都编入了 kernel，所以用户在使用前无需使用 `insmod` 来加载模块。

```

--- USB support
<*> Support for Host-side USB
[*] USB device filesystem
[*] USB device class-devices (DEPRECATED)
-* USB selective suspend/resume and wakeup
[*] EHCI HCD (USB 2.0) support
[*] Root Hub Transaction Translators
    Select transceiver (On-chip (FL/LS only)) --->
<*> USB Mass Storage support

```

在 Linux 启动完成后，用户可将 U 盘插入 USB 插口，接着系统会自动枚举 USB 设备，并加载其相关的文件系统，显示信息如下：

```
scsi 0:0:0:0: Direct-Access    Generic   USB 2.0           2.40 PQ: 0 ANSI: 2
sd 0:0:0:0: [sda] 247808 512-byte hardware sectors: (126 MB/121 MiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] 247808 512-byte hardware sectors: (126 MB/121 MiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Assuming drive cache: write through
sda:
sd 0:0:0:0: [sda] Attached SCSI removable disk
sd 0:0:0:0: Attached scsi generic sg0 type 0
```

系统加载 USB 文件系统至/mnt/sda 目录下，用户进入该目录后，即可使用 ls, cat 等命令显示文件列表和文件内容。

```
[root@M54418TWR /]# cd /mnt/sda
[root@M54418TWR sda]# ls
u-boot_mram.bin  u-boot_nand.bin  u-boot_sbf.bin  uboot_linux_modelo.txt
```

## 2.2.7 IrDA 驱动

标准的 Linux BSP 不带 IrDA 驱动，这里的 IrDA 驱动是在加上补丁之后开启的驱动配置。红外的驱动使能也在内核的配置菜单中选择。

```
--Linux Kernel Configuration--
Device Drivers ->Character devices ->Serial drivers ->
[*] ColdFire IRDA support
```

由于在本设计中，系统 UART6 外接红外接口，所以使能 UART6。具体见串口驱动配置一章。需要注意的是，在内核原码目录 drivers/serial/下的 mcf.c 中定义了一个宏 SERIAL\_IRDA\_LINE，该数值应与红外串口在/dev 下对应的设备文件 ttySn 的序号保持一致，该值的默认值为 4，即如果使用 ttyS4 作为红外接口，则定义 SERIAL\_IRDA\_LINE 为 4，以此类推。

- IrDA 驱动

IrDA 的驱动在打上补丁后的 drivers/serial/mcf.c 中。使能 IrDA 的宏为 CONFIG\_SERIAL\_COLDFIRE\_IRDA。在 mcf\_startup()中初始化 PWM，来开启 38KHz 的调制波形。其他的就按照正常的串口发送程序一样即可。

- IrDA 应用样例

下面是一个简单的自循环读取红外接口数据的例子，使用了上面 UART 驱动一节中所定义的一些函数和头文件

```
int main(int argc, char *argv[]){
    int fd;
    int readbyte,writebyte;
    char buff[512];
    char *dev = "/dev/ttyS4"; /* 串口二 */
    char *bdr;
    int ret;

    fd = OpenDev(dev);/*打开文件*/
    set_speed(fd,300);/*红外调制频率为 38KHz,这里选择了 300BPS 保证调制的正确*/
    if (set_Parity(fd,8,1,'N') == FALSE) {
        printf("Set Parity Error\n");
        exit (FALSE);
    }
    printf("IrDa: Please input chars and it will be looped back through IrDa port!!\n");
    while(1){
        buff[0] = getchar();
        if (buff[0] != '\n'){/*去除换行符*/
            writebyte = write(fd,buff,1);/*发送数据*/
            readbyte = read(fd, &buff[1], 1);/*接收数据*/
            printf("\nread bytes: %d\n",readbyte);/*打印接收结果*/
            printf("Received Data:%c\n",buff[1]);
        }
    }
}
```

```

}
close(fd);
exit (TRUE);
}

```

## 2.2.8 LED 和 GPI 驱动（扩展 GPIO 驱动）

参考设计的 LED 是通过总线扩展的 GPO 模块来驱动的，GPI 模块类似。因此对于 LED 的控制可以通过对系统的一个区域写操作进行。请参考硬件手册的 GPIO 一节了解详细的硬件配置。

- 扩展 GPIO 驱动说明

要启用扩展的 GPIO 驱动，需要在内核的设备驱动

```

--Linux Kernel Configuration--
Device Drivers → Concentrator Demo Drivers →
[*] Extend GPIO support

```

系统中定义了片选 CS4 的基地址为 0x2000\_0000

```
#define FB_CS4_BA 0x20000000
```

在 LED 驱动的初始化部分主要用来先申请设备 ID，然后对片选 4 的空间进行初始化配置

```

MCF_GPIO_PAR_CS |= MCF_GPIO_PAR_CS_CS4_CS4;
MCF_FBCS_CSAR(4) = FB_CS4_BA;
MCF_FBCS_CSMR(4) = MCF_FBCS_CSMR_BAM_64K | MCF_FBCS_CSMR_V;
MCF_FBCS_CSCR(4) = MCF_FBCS_CSCR_ASET(01) |
MCF_FBCS_CSCR_RDAH(0b11) |
MCF_FBCS_CSCR_WRAH(0b11) |
MCF_FBCS_CSCR_WS(0x18) |
MCF_FBCS_CSCR_AA |
MCF_FBCS_CSCR_PS(01);

```

```

ConcentratorExtLED = ioremap(FB_CS4_BA,0x100); //remap the phy address to virtual address
ConcentratorExtSTATUS = ConcentratorExtLED + 0x8;

```

ConcentratorExtLED 就是 LED 的基地址，而 ConcentratorExtSTATUS 则是 GPI 的基地址。这样只在驱动中对这 2 个基地址的宏定义进行操作即可对 GPO 和 GPI 控制。

```

#define CONCENTRATOR_LED MCF_REG08(ConcentratorExtLED)
#define CONCENTRATOR_STATUS MCF_REG08(ConcentratorExtSTATUS)

```

LED 的驱动通过 ioctl 函数来对 LED 进行控制。对每个 LED 的 ON 和 OFF 都有对应的宏来操作。

```

static int ConcentratorExtGPIO_ioctl(struct inode *inodep, struct file *filp, unsigned int cmd, unsigned long arg)
{
    switch(cmd)
    {
        case CMD_LED_POWAVABL_ON:
            LED_status &= (~LED_POWAVABL);
            CONCENTRATOR_LED = LED_status;
            break;
        case CMD_LED_POWAVABL_OFF:
            LED_status |= (LED_POWAVABL);
            CONCENTRATOR_LED = LED_status;
            break;
        case CMD_LED_NOPOW_ON:
            LED_status &= (~LED_NOPOW);
            CONCENTRATOR_LED = LED_status;
            break;
        case CMD_LED_NOPOW_OFF:
            LED_status |= (LED_NOPOW);
            CONCENTRATOR_LED = LED_status;
            break;
        case CMD_LED_METRD_ON:
            LED_status &= (~LED_METRD);
            CONCENTRATOR_LED = LED_status;
            break;
        case CMD_LED_METRD_OFF:
            LED_status |= (LED_METRD);
            CONCENTRATOR_LED = LED_status;
            break;
        case CMD_LED_485CHAIN_ON:
            LED_status &= (~LED_485CHAIN);
            CONCENTRATOR_LED = LED_status;

```

```

        break;
    case CMD_LED_485CHAIN_OFF:
        LED_status |= (LED_485CHAIN);
        CONCENTRATOR_LED = LED_status;
        break;
    case CMD_LED_POWER_ON:
        LED_status &= (~LED_POWER);
        CONCENTRATOR_LED = LED_status;
        break;
    case CMD_LED_POWER_OFF:
        LED_status |= (LED_POWER);
        CONCENTRATOR_LED = LED_status;
        break;

    case CMD_LED_ALERT_ON:
        LED_status &= (~LED_ALERT);
        CONCENTRATOR_LED = LED_status;
        break;

    case CMD_LED_ALERT_OFF:
        LED_status |= (LED_ALERT);
        CONCENTRATOR_LED = LED_status;
        //printk(KERN_NOTICE "CMD_LED_ALERT_OFF, LED_status = 0x%x\n",LED_status);
        break;
    case CMD_LED_ALL_OFF:
        LED_status = 0xFF;
        CONCENTRATOR_LED = LED_status;
        //printk(KERN_NOTICE "CMD_LED_ALL_OFF, LED_status = 0x%x\n",LED_status);
        break;
    case CMD_LED_ALL_ON:
        LED_status = 0x0;
        CONCENTRATOR_LED = LED_status;
        //printk(KERN_NOTICE "CMD_LED_ALL_ON, LED_status = 0x%x\n",LED_status);
        break;
    default:
        printk(KERN_NOTICE "Concentrator Error command\n");
        return -1;
        break;
    }
    return 0;
}

```

而读 GPI 状态的函数则更简单，直接读取 GPI 的宏即可

```

static ssize_t ConcentratorExtGPIO_read(struct file *filp, char __user *buffer, size_t count, loff_t *ppos)
{
    unsigned char status;
    status = CONCENTRATOR_STATUS;
    copy_to_user(buffer, &status, sizeof(char));
    return sizeof(char);
}

```

#### ● 使用 LED 驱动

在应用程序使用时，首先系统应该先加载 LED 驱动，推荐在系统启动时自动加载。在参考平台发布时，已经将 LED 驱动自动加载了。对应系统的/dev/Concentrator\_ExtGPIO 设备。使用时先打开文件，然后就可以进行 LED 控制了。

```

extgpio_fd = open("/dev/Concentrator_ExtGPIO", 0);
if (extgpio_fd < 0)
{
    perror("cann't open device /dev/Concentrator_ExtGPIO \n");
    goto label_extGPIO_APP;//exit(1);
}
...
ioctl(extgpio_fd, CMD_LED_ALL_OFF, 0);
ioctl(extgpio_fd, CMD_LED_POWER_ON, 0);
...

```

同样可以读取文件来读出 GPI 状态。

## 2.2.9 LCD 驱动

参考平台上集成的 LCD 模块采用 160x160 的点阵屏，驱动模块为 Sitronix 公司的 LCDC ST7529

的单色 LCD 模组。LCD 驱动实现了在 LCD 屏上的 GB2312 字符等显示。

- LCD 驱动配置

要使能 LCD 驱动，需要在内核的设备驱动

--Linux Kernel Configuration--

Device Drivers → Concentrator Demo Drivers → ST7529 LCDC support (CONFIG\_ST7529\_LCDC)

- LCD 驱动设计介绍

由于集中器的显示应用功能较为简单，本驱动并未采用为 Frame Buffer 类型驱动，而采用了支持清屏、画点、画水平或竖直线、画矩形、反色、显示中文 GB2312 和英文混合文字、画简单 bitmap、开关背光功能。

LCDC 显示 buffer 位于 LCDC 模块内部，其接口是 8080 总线接口。因此，我们将此模块直接挂载到 MCF5441x 的 Flexbus 总线上，通过 FB\_ADO 选择访问 LCDC 显示 buffer 或 LCDC 寄存器。原理图详见参考平台硬件手册。

- LCD 驱动使用

ioctl 支持以下命令：

ST7529LCDC\_BACKLIGHT\_ON：开启背光

ST7529LCDC\_BACKLIGHT\_OFF：关闭背光

ST7529LCDC\_DISPLAY\_ON：开启 LCD 显示

ST7529LCDC\_DISPLAY\_OFF：关闭 LCD 显示

ST7529LCDC\_CLEAR\_SCREEN：清屏

ST7529LCDC\_CLEAR\_AREA：清除区域

ST7529LCDC\_INVERT\_AREA：区域反色操作

ST7529LCDC\_SET\_PIXEL：画点

ST7529LCDC\_DRAW\_LINE：画线（只支持水平或竖线）

ST7529LCDC\_DRAW\_RECT：画矩形

ST7529LCDC\_DRAW\_TEXT：显示中文 GB2312 和英文混合文字

ST7529LCDC\_DRAW\_BITMAP：画简单 bitmap

其中前 5 个命令无需额外参数，调用 ioctl 时只需将参数部分写 0 即可。

使用其他命令则需要填写相应参数：

如画矩形，`st7529_rect_t rect = {0,0,159,159,0,BLACK};` `ioctl(lcdc_fd, ST7529LCDC_DRAW_RECT &rect);`

其他命令使用，可参考 ERTU\_APP。用户可以参考驱动和应用程序代码针对自己所使用的 LCD 模块进行修改和开发。

注意：对于中文显示，目前 LCD 驱动只支持 GB2312，所以在编写程序时，注意设置代码编辑器的中文编码方式。特别是在 linux 下的编辑器，其通常的默认中文编码方式是 unicode(UTF-8)。显示 UTF-8 编码方式的中文将会是乱码。

## 2.2.10 Keypad 驱动

参考平台的按键采用中断扫描方式，其硬件连接请参考集中器参考平台硬件手册。在初始化时，先将各 GPIO 配置成输出低电平状态，相当于按键一端接地。此时如果有按键按下，则触发 CPU 中断 IRQ2。进入中断服务程序后，关闭中断防止重入，将 IRQ2 配置成 GPIO 输出低电平，而另外一端的所有 GPIO 配置成输入，读取状态，此时只有被按下的按键的状态被读取为低，其他的 GPIO 则为高。在中断服务程序中启动按键去抖状态机，首先定时去抖

10ms, 期间按键的状态设置为不定态 KEYSTATUS\_DOWNX。在 10ms 到时后, 系统自动调用定时回调函数, 在回调函数中判断, 若为不定态, 表示按键已经去抖 10ms, 进入稳定的按键按下状态 KEYSTATUS\_DOWN, 再启动新的定时器, 定时 100ms, 此后每隔 100ms 定时并在回调函数中检测按键状态, 直到检测到按键抬起, 则重新复位状态机, 并开启按键中断, 等待下一次按键按下触发。

- 按键驱动

要使得按键驱动, 需要在内核的设备驱动选择

```
--Linux Kernel Configuration--
Device Drivers → Concentrator Demo Drivers →
[*] Keypad Support
```

驱动的初始化主要配置好各 GPIO, 并注册 IRQ2 的中断服务程序, 初始化去抖定时器

```
//Enable the GPIO pull up resistor
MCF_GPIO_PCR_H |= 0xC;
MCF_GPIO_PCR_I |= 0xFF0;

//clear the I output as 0
MCF_GPIO_PCLRR_I = 0x3;
MCF_GPIO_PCLRR_H = 0xFD;

//set as output
MCF_GPIO_PDDR_I |= 0xFC;
MCF_GPIO_PDDR_H |= 0x2;

//assign related pin as GPIO
MCF_GPIO_PAR_DBGH1 = 0;
MCF_GPIO_PAR_DBGH0 = 0;

//Following code setup the edgeport
//The keypad matrix IRQ is using edgeport2(IRQ2)
MCF_GPIO_PAR_IRQOL |= 0xC;
MCF_GPIO_HCR1 |= 0x20; //enable the IRQ2 Hysteresis Control
//register the interrupt handle
request_irq(KEYPAD_IRQ, ConcentratorKeypad_irq, IRQF_DISABLED, DRIVER_NAME, (void*)ConcentratorKeypad_dev);

//set the interrupt level
MCF_INTCO_ICR2 = 2; //interrupt level 2
//Enable the IRQ
MCF_EPORT_EPIER |= MCF_EPORT_EPIER_EPIE2;
//Clear the Eport flag, write 1 to clear the bit
MCF_EPORT_EPFR |= MCF_EPORT_EPFR_EPF2;
//falling edge trigger
MCF_EPORT_EPPAR &= 0xFFCF;
MCF_EPORT_EPPAR |= MCF_EPORT_EPPAR_EPPA2_FALLING;

init_waitqueue_head(&q);
ConcentratorKeypad_waitflag = 0;
//init timer
key_timer.function = key_timer_callback;
key_timer.data = 0;
init_timer(&key_timer);
key_status = KEYSTATUS_UP;
```

按键的中断服务程序 ConcentratorKeypad\_irq 是在按键被按下的时候触发, 然后记录按键状态, 并启动去抖的状态机。用户可以按照前面的扫描说明来了解代码。

```
static irqreturn_t ConcentratorKeypad_irq(int irq, void *dev)
{
    spin_lock(&lock);

    //Disable the interrupt temporarily
    MCF_EPORT_EPIER &= (~MCF_EPORT_EPIER_EPIE2);

    //*****check which key is pressed*****//
    //Set the Debug GPIO pin as input
    MCF_GPIO_PDDR_I &= 0x3;
    MCF_GPIO_PDDR_H &= 0xFD;
    MCF_GPIO_PPDSR_I |= 0xFC;
    MCF_GPIO_PPDSR_H |= 0x2;
    //set IRQ2 as Output 0
    MCF_GPIO_PCLRR_C = 0xFB;
```

```

MCF_GPIO_PDDR_C |= 0x4;
MCF_GPIO_PAR_IRQOL &= 0xF3;

//Now use the Debug GPIO to detect which key is pressed
keycode = MCF_GPIO_PPDSDR_H & 0x2;
keycode |= (MCF_GPIO_PPDSDR_I & 0xFC);

if(keycode == 0xFE)
{
    //printf(KERN_NOTICE"bad keypad irq keycode = 0x%x key_staus = %d\n", keycode,key_status);
    //recover the GPIO and IRQ
    MCF_GPIO_PDDR_C &= 0xFB; //set IRQ2 as input first , to release the output 0

    //The keypad matrix IRQ is using edgeport2(IRQ2)
    MCF_GPIO_PAR_IRQOL |= 0xC;

    //clear the I/H GPIO output as 0
    MCF_GPIO_PCLRR_I = 0x3;
    MCF_GPIO_PCLRR_H = 0xFD;

    //set as output
    MCF_GPIO_PDDR_I |= 0xFC;
    MCF_GPIO_PDDR_H |= 0x2;
    //falling edge trigger
    MCF_EPORT_EPPAR &= 0xFFCF;
    MCF_EPORT_EPPAR |= MCF_EPORT_EPPAR_EPPA2_FALLING;
    //reenable
    MCF_EPORT_EPFR |= MCF_EPORT_EPFR_EPF2;

    MCF_EPORT_EPIER |= MCF_EPORT_EPIER_EPIE2;
    return IRQ_NONE;
}
key_status = KEYSTATUS_DOWNX;
key_timer.expires = jiffies + KEY_TIME_DELAY_10MS;
add_timer(&key_timer);

spin_unlock(&lock);
return IRQ_HANDLED;
}

```

回调函数用于实现去抖状态机的整个过程。

```

static void key_timer_callback(unsigned long data)
{
    keycode = MCF_GPIO_PPDSDR_H & 0x2;
    keycode |= (MCF_GPIO_PPDSDR_I & 0xFC);
    if(keycode != 0xFE)
    { //some key pressed
        if(key_status == KEYSTATUS_DOWNX) //already delayed 10ms,jitter finished
        {
            switch(keycode)
            {
                case 0xDE:
                    printk(KERN_INFO "UP Key\n"); //Up Key
                    break;
                case 0x7E:
                    printk(KERN_INFO "DOWN Key\n"); //DOWN Key
                    break;
                case 0xFC:
                    printk(KERN_INFO "LEFT Key\n"); //LEFT Key
                    break;
                case 0xFA:
                    printk(KERN_INFO "RIGHT Key\n"); //RIGHT Key
                    break;
                case 0xBE:
                    printk(KERN_INFO "CANCEL Key\n"); //CANCEL Key
                    break;
                case 0xEE:
                    printk(KERN_INFO "OK Key\n"); //OK Key
                    break;
                case 0xF6:
                    printk(KERN_INFO "RESERVE Key\n"); //RESERVE Key
                    break;
                default:
                    keycode = NONE_KEY;
                    break;
            }
        }
    }
}

```

```

    key_status = KEYSTATUS_DOWN;
    key_timer.expires = jiffies + KEY_TIME_DELAY_100MS; //delay for keypad rising detect
    if(keycode != NONE_KEY) //if there is a valid key pressed, wake up the waiting queue
    {
        ConcentratorKeypad_waitflag = 1;
        wake_up_interruptible(&(ConcentratorKeypad_wq));
    }
    add_timer(&key_timer);
}
else
{
    key_timer.expires = jiffies + KEY_TIME_DELAY_100MS; //delay for keypad rising detect
    add_timer(&key_timer);
}
}
else
{//the keypad rise, and no keypad pressed
    key_status = KEYSTATUS_UP;
    //recover the GPIO and IRQ
    MCF_GPIO_PDDR_C &= 0xFB; //set IRQ2 as input first , to release the output 0

    //The keypad matrix IRQ is using edgeport2(IRQ2)
    MCF_GPIO_PAR_IRQOL |= 0xC;

    //clear the I/H GPIO output as 0
    MCF_GPIO_PCLRR_I = 0x3;
    MCF_GPIO_PCLRR_H = 0xFD;

    //set as output
    MCF_GPIO_PDDR_I |= 0xFC;
    MCF_GPIO_PDDR_H |= 0x2;
    //falling edge trigger
    MCF_EPORT_EPPAR &= 0xFFCF;
    MCF_EPORT_EPPAR |= MCF_EPORT_EPPAR_EPPA2_FALLING;
    //reenable
    MCF_EPORT_EPFR |= MCF_EPORT_EPFR_EPF2;

    MCF_EPORT_EPIER |= MCF_EPORT_EPIER_EPIE2;
}
}
}

```

由于按键的驱动可以支持 **block** 和 **nonblock** 两种方式读取。使得应用程序在读取键值时可以被自动挂起，或者继续运行。如果采用自动挂起方式，则也需要按键驱动支持唤醒，这就是在中断服务程序中为什么需要采用一个等待队列并唤醒的机制。**block** 和 **nonblock** 通过打开文件的方式来选择指定。

按键驱动的读函数 **ConcentratorKeypad\_read** 是它和应用程序的主要通道。用来传递按键的键值。当读取的时候如果有按键键值按下，则传递给应用程序。如果没有则判断驱动文件是否为 **block** 方式。如果驱动采用 **block** 方式打开，则将应用程序挂起，直到有按键按下时唤醒继续执行。如果是 **nonblock** 方式打开，则直接返回无效键值。

```

static ssize_t ConcentratorKeypad_read(struct file *filp, char __user *buffer, size_t count, loff_t *ppos)
{
    unsigned int keypad_ret = NONE_KEY;
retry:
    keypad_ret = KeypadRead();
    if(keypad_ret != NONE_KEY)
    {
        copy_to_user(buffer, &keypad_ret, sizeof(int));
        printk(KERN_INFO "keypad_ret is 0x%x\n", keypad_ret);

        return sizeof(int);
    }
    else //no key pressed
    {
        if(filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        wait_event_interruptible(ConcentratorKeypad_wq, ConcentratorKeypad_waitflag);
        ConcentratorKeypad_waitflag = 0;

        if(signal_pending(current))
        {
            //be waked up by signal
            printk("return -ERESTARTSYS\n"); //tell FS layer to handle
            return -ERESTARTSYS; //tell the system call that this function should be call again
        }
    }
}

```

```

    }
    goto retry;
}

return sizeof(int);
}

```

- 应用程序使用按键驱动

参考平台在系统启动时自动加载按键驱动，设备为/dev/Concentrator\_Keypad。在使用时只需打开驱动文件读取即可。

```

int main(void)
{
    int keypad_fd;
    int key_value;
    char* s;
    keypad_fd = open("/dev/Concentrator_Keypad", 0);//O_NONBLOCK;//block or nonblock
    if (keypad_fd < 0) {
        perror("cann't open device /dev/Concentrator_Keypad");
        exit(1);
    }
    while(1)
    {
        int ret = read(keypad_fd, &key_value, sizeof key_value);
        //printf("read return 0x%x , EAGAIN = 0x%x\n",ret,-EAGAIN);
        if(ret != sizeof(key_value))
            key_value = 0;
        switch(key_value)
        {
            case UP_KEY:    s = "UP";    break;
            case DOWN_KEY:  s = "DOWN";  break;
            case LEFT_KEY:  s = "LEFT";  break;
            case RIGHT_KEY: s = "RIGHT"; break;
            case CANCEL_KEY: s = "CANCEL"; break;
            case OK_KEY:    s = "OK";    break;
            case REV_KEY:   s = "RESERVE"; break;
            default: s = "NONE";break;
        }
        if(key_value != 0)
            printf("You pressed buttons %s\n", s);
    }
    close(keypad_fd);
    return 0;
}

```

## 2.2.11 添加外部中断

MCF5441x 拥有 5 个外部中断，其中 2 个已经被使用。用户还可以使用剩下的 3 个扩展额外的功能。本节以 IRQ1 为例介绍在 Linux 环境下，如何添加相应的外部中断驱动。

- 中断驱动

样例代码：test\_irq1.c

```

#include <linux/module.h>
#include <linux/init.h>
#include <linux/ioport.h>
#include <linux/interrupt.h>
#include <linux/delay.h>
#include <linux/highmem.h>
#include <linux/dma-mapping.h>
#include <linux/scatterlist.h>
#include <linux/uaccess.h>
#include <linux/irq.h>
#include <linux/io.h>

#include <linux/platform_device.h>
#include <asm/coldfire.h>
#include <asm/mcfsim.h>

#define DRIVER_NAME "test_irq1"

#define INTERRUPT_IRQ1_VECTOR_NUM 65 //IRQ1 中断号

```

```

static int major = 0; /* 动态分配 major 号 */
module_param(major, int, 0);
static struct class *test_irq1_class;
#define TEST_IRQ1_DEBUG 1
#ifdef TEST_IRQ1_DEBUG
#define DBG(fmt, args...) printk(KERN_INFO "[%s] " fmt "\n", __func__, ## args)
#else
#define DBG(fmt, args...) do {} while (0)
#endif

static spinlock_t lock; //自旋锁
//中断处理程序
static irqreturn_t test_irq1_detect_irq(int irq, void *dev_id)
{
    spin_lock(&lock);
    MCF_EPORTEPIER = MCF_EPORTEPIER & (~MCF_EPORTEPIER_EPIE1);
    MCF_EPORTEPF1 = MCF_EPORTEPF1|MCF_EPORTEPF1_EPF1;
    printk("IRQ1 happens!\r\n");
    DBG("MCF_EPORTEPPAR = 0x%x, MCF_EPORTEPF1 = 0x%x, MCF_INTCO_IPRL = 0x%x\r\n",
        MCF_EPORTEPPAR, MCF_EPORTEPF1, MCF_INTCO_IPRL);
    MCF_EPORTEPPAR &= 0xffff3;
    MCF_EPORTEPPAR = MCF_EPORTEPPAR | MCF_EPORTEPPAR_EPPA1_FALLING;
    MCF_EPORTEPF1 = MCF_EPORTEPF1|MCF_EPORTEPF1_EPF1;
    MCF_EPORTEPIER = MCF_EPORTEPIER|MCF_EPORTEPIER_EPIE1;
    spin_unlock(&lock);
    return 0;
}

//以下为文件操作接口定义
static int test_irq1_open(struct inode *inode, struct file *filp)
{
    return 0;
}
static int test_irq1_close(struct inode *inode, struct file *filp)
{
    return 0;
}
//ioctl, 可用于设置 irq 类型: Falling/Rising edge 或 level 触发
static int test_irq1_ioctl(struct device *dev, unsigned int cmd, unsigned long arg)
{
    return 0;
}
static struct file_operations test_irq1_fops = {
    .open = test_irq1_open,
    .close = test_irq1_close,
    .ioctl = test_irq1_ioctl,
    .owner = THIS_MODULE
};

//以下为基本 driver 接口定义
//probe: 当本模块被加载, 即 test_irq1_drv_init() 被运行过之后, 系统会调用该函数, 继而创建/dev/test_irq1
//从而可以用 open("/dev/test_irq1", NULL) 打开该模块驱动。
static int test_irq1_probe(struct platform_device *pdev)
{
    int ret;
    struct device *dev;
    dev_t devt;

    printk(KERN_INFO "%s: start probe\n", __func__);
    MCF_GPIO_SRQR_IRQ0 = 3;
    MCF_GPIO_PAR_IRQ0H |= MCF_GPIO_PAR_IRQH_IRQ1;
    MCF_EPORTEPPAR &= 0xffff3;
    MCF_EPORTEPPAR = MCF_EPORTEPPAR | MCF_EPORTEPPAR_EPPA1_FALLING;
    MCF_EPORTEPIER = MCF_EPORTEPIER | MCF_EPORTEPIER_EPIE1;
    MCF_INTCO_ICR1 = 3; /* IRQ1 */
    ret = request_irq(INTERRUPT_IRQ1_VECTOR_NUM, test_irq1_detect_irq, IRQF_DISABLED,
        pdev->name, pdev); //申请中断
    if (ret) {
        printk(KERN_INFO "%s: request irq fail %x\n", __func__, ret);
    }
    spin_lock_init(&lock);
    devt = MKDEV(major, 0);
    dev = device_create(test_irq1_class, &pdev->dev, devt, NULL, DRIVER_NAME); //创建/dev/irq_test1
    ret = IS_ERR(dev) ? PTR_ERR(dev) : 0;
    if (ret)
        {printk(KERN_ERR "unable to create %s class\n", DRIVER_NAME);}
    return ret;
}
//注销该驱动, 删除/dev/test_irq1
static int test_irq1_remove(struct platform_device *pdev)
{
    dev_t devt;

```

```

    DBG("%s: remove\n", __func__);
    devt = MKDEV(major, 0);
    device_destroy(test_irq1_class, devt);
    MCF_EP0RT_EPIER = MCF_EP0RT_EPIER & (~MCF_EP0RT_EPIER_EPIE1);
    MCF_EP0RT_EPFR = MCF_EP0RT_EPFR | MCF_EP0RT_EPFR_EPF1;
    free_irq(INTERRUPT_IRQ1_VECTOR_NUM, pdev);
    return 0;
}

/*-----*/
static struct platform_driver test_irq1_driver = {
    .probe = test_irq1_probe,
    .remove = test_irq1_remove,
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
    },
};

static void test_irq1_release(struct device *dev)
{
    return;
}
static struct platform_device test_irq1_device = {
    .name = DRIVER_NAME,
    .id = -1,
    .dev = { .release = test_irq1_release, },
};

/*****
 * Driver init/exit
 *****/

static int __init test_irq1_drv_init(void)
{
    int ret;
    printk(KERN_INFO DRIVER_NAME
           ": Freescale IRQ1 Test Program driver init\r\n");
    ret = register_chrdev(major, DRIVER_NAME, &test_irq1_fops);
    if (ret < 0) {
        printk(KERN_INFO "st7529lcdc: can't get major number\n");
        return ret;
    }
    if (major == 0) {
        major = ret; /* dynamic */
    }
    test_irq1_class = class_create(THIS_MODULE, "test_irq1");
    if (IS_ERR(test_irq1_class)) {
        unregister_chrdev(major, DRIVER_NAME);
        return PTR_ERR(test_irq1_class);
    }
    ret = platform_device_register(&test_irq1_device);
    if (ret)
    {
        printk(KERN_INFO DRIVER_NAME ": register device failed! ret = %d\r\n", ret);
        unregister_chrdev(major, DRIVER_NAME);
        goto exit;
    }
    ret = platform_driver_register(&test_irq1_driver);
    if (ret)
    {
        printk(KERN_INFO DRIVER_NAME
           ": register driver failed! ret = %d\r\n", ret);
        platform_device_unregister(&test_irq1_device);
        unregister_chrdev(major, DRIVER_NAME);
        goto exit;
    }
exit:
    return ret;
}

static void __exit test_irq1_drv_exit(void)
{
    printk(KERN_INFO DRIVER_NAME
           ": Freescale IRQ1 Test Program driver exit\r\n");
    class_destroy(test_irq1_class);
    unregister_chrdev(major, DRIVER_NAME);
    platform_device_unregister(&test_irq1_device);
    platform_driver_unregister(&test_irq1_driver);
}

```

```
}  
  
module_init(test_irq1_drv_init);  
module_exit(test_irq1_drv_exit);  
  
MODULE_AUTHOR("Ju Yingyi<b19829@freescale.com>");  
MODULE_DESCRIPTION("Freescale MCF5441x IRQ1 Test Program");  
MODULE_LICENSE("GPL");
```

编译的 Make 文件可以如下：

```
obj-m := test_irq1.o  
#KDIR 是 kernel 源代码目录，请根据此 Makefile 位置更改 KDIR  
KDIR := ../..  
PWD := $(shell pwd)
```

```
default:  
    make -C $(KDIR) M=$(PWD) modules
```

```
clean:  
    rm -rf *.o *.cmd *.ko *.mod.c .tmp_versions
```

- 测试中断

打开 console，转到 Makefile 和 test\_irq1.c 目录下，运行“make ARCH=m68k CROSS\_COMPILE=/opt/freescale/usr/local/gcc-4.4.54-eglibc-2.10.54/m68k-linux/bin/m68k-linux-gnu-”，生成 test\_irq1.ko，将它拷贝到 NFS rootfs/usr 目录下。NFS 方式启动目标板后，运行“insmod /usr/test\_irq1.ko”：

```
[root@M54418TWR usr]# insmod test_irq1.ko  
test_irq1: Freescale IRQ1 Test Program driver init  
test_irq1_probe: start probe
```

检查/dev 目录下是否已建立 test\_irq1 设备：

```
[root@M54418TWR /]# ls /dev/test_irq1 -l  
crw-rw---- 1 root root 248, 0 Jun 18 12:00 /dev/test_irq1
```

测试 irq1，外部触发 IRQ1 中断：

```
IRQ1 happens!
```

```
IRQ1 happens!
```

注销驱动：

```
# rmmod test_irq1  
test_irq1: Freescale IRQ1 Test Program driver exit  
[test_irq1_remove] test_irq1_remove: remove
```

## 2.2.12 看门狗模块

参考平台硬件支持内部看门狗以及外部看门狗两种选择。内部看门狗的驱动请参考 BSP 的文档相关驱动章节。外部看门狗的驱动只需要采用 2 个 GPIO 来控制使能和喂狗操作，暂时还没有编写，用户可以按照硬件连接自行编写。

## 2.2.13 GPIO 驱动

在参考平台中，有许多的 CPU 的 GPIO 用来控制或读取各接口-本地通信模块/远程通信模块。参考平台的软件没有一一给出所有的例子。但是用户可以很容易的参考 2.2.10 按键驱动中控制 GPIO 的写法开发每个/组 GPIO 的驱动。而对于通过总线扩展方式扩展的 GPIO 则可以参考 2.2.8 节的例子。