



浙江师范大学  
ZHEJIANG NORMAL UNIVERSITY

# 嵌入式系统 实验指导书

计算机科学与技术学院

## 目录

|                            |    |
|----------------------------|----|
| 实验一 Linux 系统移植及编译环境构建..... | 3  |
| 实验二 LED 与按键驱动实验.....       | 17 |
| 实验三 LCD 与触摸屏实验 .....       | 35 |
| 实验四 QT 移植与开发实验 .....       | 60 |
| 实验五 SQLite 数据库实验.....      | 75 |
| 实验六 网络服务器 BOA 实验.....      | 78 |

## 实验一 Linux 系统移植及编译环境构建

### 一、实验目的

本实验主要涉及嵌入式开发环境设置和嵌入式 Linux 操作系统的移植，包括:构建系统编译环境、编译脚本分析、U-Boot 移植、Linux 内核移植和根文件系统构建。

### 二、实验环境

- 硬件：电脑（推荐：主频 2GHz+，内存：1GB+）s6818 系列实验平台；
- 软件：ZEmberOS 操作系统。

### 三、实验内容一：构建系统编译环境

交叉编译是嵌入式开发过程中的一项重要技术，简单地说，就是在一个平台上生成另一个平台上的可执行代码。交叉编译这个概念的出现和流行是和嵌入式系统的广泛发展同步的。我们常用的计算机软件，都需要通过编译的方式，把使用高级计算机语言编写的代码（比如 C 代码）编译（compile）成计算机可以识别和执行的二进制代码。交叉编译的主要特征是某机器中执行的程序代码不是由本机编译生成，而是由另一台机器编译生成，一般把前者称为目标机，后者称为宿主机，如图 1-1 所示。这是因为目标平台上不允许或不能够安装开发所需要的编译器，而又需要该编译器的某些特征；有时是因为目标平台上的资源贫乏，无法运行开发所需要的编译器；有时是因为目标平台还没有建立，连操作系统都没有，根本谈不上运行编译器。

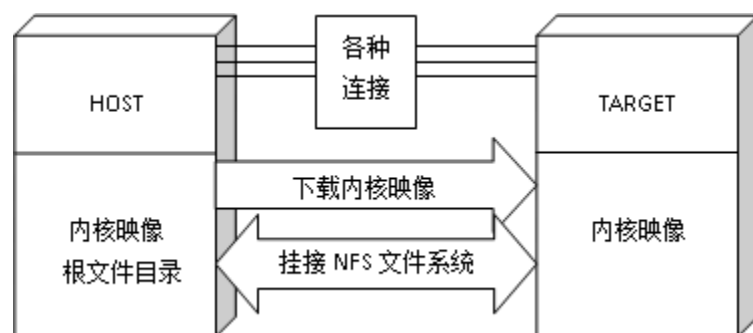


图 1-1 交叉开发模型

交叉编译的引入主要是由于不同架构的 CPU 的指令集不相同，比如在 X86 架构的处理器上编译运行的程序，不能直接在 XScale 构架处理器上运行，不同的 CPU 有不同的编译器；另一方面，编译器本身也是程序，也要在某一个 CPU 平台上运行，而嵌入式目标系统不能提供足够的资源，不能运行编译器。

这里所谓的平台，实际上包含两个概念：体系结构（Architecture）和操作系统（Operating System）。同一个体系结构可以运行不同的操作系统，同样，同一操作系统也可以在不同的体系结构上运行。例如，常说的 X86 Linux 平台实际上是 Intel X86 体系结构和 Linux for X86 操作系统的统称，而 X86 WinNT 平台实际上是 Intel X86 体系结构和 Windows NT for X86 操作系统的统称。

要进行交叉编译，我们需要在主机平台上安装对应的交叉编译工具链（cross compilation tool chain），然后用这个交叉编译工具链编译我们的源代码，最终生成可在目标平台上运行的代码。

运行于宿主机上的交叉开发环境至少必须包含编译调试模块，其编译器为交叉编译器。宿主机一般为基于 X86 体系的台式计算机，而编译出的代码必须在 ARM 体系结构的目标机上运行，这就是所谓的交叉编译。在宿主机上编译好目标代码后，通过宿主机到目标机的调试通道将代码下载到目标机，然后由运行于宿主机的调试软件控制代码在目标机上运行调试。为了方便调试开发，交叉开发环境一般为一个整合编辑、编译汇编链接、调试、工程管理及函数库等功能模块的集成开发环境（Integrated Development Environment，IDE）。

#### 四、实验步骤一：构建系统编译环境

s6818 系列实验平台对应的系统代码及编译工具位置在：DISK-Android-s6818\03-系统代码，用户在使用前需要将光盘内文件（x6818.tar.bz2、arm-2009q3.tar.bz2）拷贝到 ubuntu 系统的/tmp 目录下：

1) 在 ubuntu 嵌入式开发环境下运行终端，执行以下命令部署 Android 系统源码：

```
$ sudo mkdir -p /usr/local/src/s6818
$ sudo chmod -R 777 /usr/local/src/s6818
$ cd /usr/local/src/s6818
$ cat /tmp/x6818.tar.bz2| tar jxv
$ tar jxvf /tmp/arm-2009q3.tar.bz2
$ ls
arm-2009q3  x6818      # 源码都部署成功
```

默认实验环境设置如下：

**工作目录:**

/usr/local/src/s6818

**源码路径:**

/usr/local/src/s6818/x6818

**应用程序交叉编译工具链路径:**

/usr/local/src/s6818/arm-2009q3/bin

**U-boot源码路径:**

/usr/local/src/s6818/x6818/uboot

**Linux内核源码路径:**

/usr/local/src/s6818/x6818/kernel

## 五、实验内容二：编译脚本分析

### 1. Shell 简介

Shell 是一种具备特殊功能的程序，它是介于使用者和 UNIX/Linux 操作系统核心程序（kernel）之间的一个接口。为什么我们说 shell 是一种介于系统核心程序与使用者之间的中介者呢？读过操作系统概论的读者们都知道操作系统是一个系统资源的管理者与分配者，当您有需求时，您得向系统提出；从操作系统的角度来看，它也必须防止使用者因为错误的操作而造成系统的伤害？众所周知，对计算机下命令得通过命令（command）或是程序（program）；程序由编译器（compiler）将程序转为二进制代码，可是命令呢？其实 shell 也是一种程序，它由输入设备读取命令，再将其转为计算机可以理解的机械码，然后执行它。

各种操作系统都有它自己的 Shell，以 DOS 为例，它的 shell 就是 command.com。如同 DOS 下有 NDOS，4DOS，DRDOS 等不同的命令解译程序可以取代标准的 command.com，UNIX 下除了 Bourne shell (/bin/sh) 外还有 C shell (/bin/csh)、Korn shell (/bin/ksh)、Bourne again shell (/bin/bash)、Tenex C shell (tcsh) ... 等其它的 shell。UNIX/Linux 将 shell 独立于核心程序之外，使得它就如同一般的应用程序，可以在不影响操作系统本身的情况下进行修改、更新版本或是添加新的功能。

在系统启动的时候，核心程序会被加载到内存，负责管理系统的工作，直到系统关闭为止。它建立并控制着处理程序，管理内存、档案系统、通讯等等。而其它的程序，包括 shell 程序，都存放在磁盘中。核心程序将它们加载到内存，执行它们，并且在它们中止后清理系统。

当您刚开始学 UNIX/Linux 系统时，您大部分的时间会花在在提示符号（prompt）下执行命令。如果您经常输入一组相同形式的命令，您可能会想要自动执行那些工作。如此，您可以将一些命令放入一个脚本文件，然后执行该脚本文件。一个 shell 脚本很像是 DOS 下的批处理文件（如 Autoexec.bat）：它把一连串的 UNIX 命令存入一个文件，然后执行该问。较成熟的脚本文件还支持若干现代程序语言的控制结构，譬如说能做条件判断、循环、脚本测试、传送参数等。要学着写脚本文件，不仅要学习程序设计的结构和技巧，而且对 UNIX/Linux

公用程序及如何运作需有深入的了解。有些公用程序的功能非常强大(例如 `grep`、`sed` 和 `awk`)，它们常被用于脚本来操控命令输出和文件。在您对那些工具和程序设计结构变得熟悉之后，您就可以开始写命令脚本。当由命令脚本执行命令时，此刻，您就已经把 `shell` 当做程序语言使用了。

## 2. Shell 的发展历史

第一个有重要意义的，标准的 UNIX `shell` 是 V7(AT&T 的第七版)UNIX，在 1979 年底被提出，且以它的创造者 Stephen Bourne 来命名。Bourne `shell` 是以 Algol 这种语言为基础来设计，主要被用来做自动化系统管理工作。虽然 Bourne `shell` 以简单和速度而受欢迎，但它缺少许多交谈性使用的特色，例如历程、别名和工作控制。

C `shell` 是在加州大学柏克来分校于 70 年代末期发展而成，而以 2BSD UNIX 的部分发行。这个 `shell` 主要是由 Bill Joy 写成，提供了一些在标准 Bourne `shell` 所看不到的额外特色。C `shell` 是以 C 程序语言作为基础，且它被用来当程序语言时，能共享类似的语法。它也提供在交谈式运用上的改进，例如命令列历程、别名和工作控制。因为 C `shell` 是在大型机器上设计出来，且增加了一些额外功能，所以 C `shell` 在小型机器上跑得较慢，即使在大型机器上跟 Bourne `shell` 比起来也显得缓慢。

有了 Bourne `shell` 和 C `shell` 之后，UNIX 使用者就有了选择，且争论哪一个 `shell` 较好。AT&T 的 David Korn 在 80 年代中期发明了 Korn `shell`，在 1986 年发行且在 1988 年成为正式的部分 SVR4 UNIX。Korn `shell` 实际上是 Bourne `shell` 的超集，且不只可在 UNIX 系统上执行，同时也可在 OS/2、VMS、和 DOS 上执行。它提供了和 Bourne `shell` 向上兼容的能力，且增加了许多在 C `shell` 上受欢迎的特色，更增加了速度和效率。Korn `shell` 已历经许多修正版，要找寻您使用的是那一个版本可在 `ksh` 提示符号下按 `Ctrl+V` 键。

三种主要的 Shell 与其分支：

在大部份的 UNIX 系统，三种著名且广被支持的 `shell` 是 Bourne `shell` (AT&T `shell`，在 Linux 下是 `BASH`)、C `shell` (Berkeley `shell`，在 Linux 下是 `TCSH`) 和 Korn `shell` (Bourne `shell` 的超集)。这三种 `shell` 在交谈 (`interactive`) 模式下的表现相当类似，但作为命令文件语言时，在语法和执行效率上就有些不同了。

Bourne `shell` 是标准的 UNIX `shell`，以前常被用来做为管理系统之用。大部份的系统管理命令脚本，例如 `rc start`、`stop` 与 `shutdown` 都是 Bourne `shell` 的脚本文件，且在单一使用者模式 (`single user mode`) 下以 `root` 登入时它常被系统管理者使用。Bourne `shell` 是由 AT&T 发展的，以简洁、快速著名。Bourne `shell` 提示符号的默认值是 `$`。

C `shell` 是柏克莱大学 (Berkeley) 所开发的，且加入了一些新特性，如命

令历程 (history)、别名 (alias)、内建算术、文件名完成 (filename completion)、和工作控制 (job control)。对于常在交谈模式下执行 shell 的使用者而言，他们较喜爱使用 C shell；但对于系统管理者而言，则较偏好以 Bourne shell 来做命令档，因为 Bourne shell 命令文件比 C shell 命令文件来的简单及快速。C shell 提示符号的默认值是 %。

Korn shell 是 Bourne shell 的超集 (superset)，由 AT&T 的 David Korn 所开发。它增加了一些特色，比 C shell 更为先进。Korn shell 的特色包括了可编辑的历程、别名、函数、正则表达式 (regular expression wildcard)、内建算术、工作控制 (job control)、合作处理 (coprocessing)、和特殊的除错功能。Bourne shell 几乎和 Korn shell 完全向上兼容 (upward compatible)，所以在 Bourne shell 下开发的程序仍能在 Korn shell 上执行。Korn shell 提示符号的默认值也是 \$。在 Linux 系统使用的 Korn shell 叫做 pdksh，它是指 Public Domain Korn Shell。

除了执行效率稍差外，Korn shell 在许多方面都比 Bourne shell 好；但是，若将 Korn shell 与 C shell 相比就很困难，因为二者在许多方面都各有所长，就效率和容易使用上看，Korn shell 是优于 C shell，相信许多使用者对于 C Shell 的执行效率都有负面的印象。

在 shell 的语法方面，Korn shell 是比较接近一般程序语言，而且它具有子程序的功能及提供较多的资料型态。至于 Bourne shell，它所拥有的资料型态是三种 shell 中最少的，仅提供字符串变量和布尔型态。在整体考量下 Korn shell 是三者中表现最佳者，其次为 C shell，最后才是 Bourne shell，但是在实际使用中仍有其它应列入考虑的因素，如速度是最重要的选择时，很可能应该采用 Bourne shell，因它是最基本的 shell，执行的速度最快。

tcsh 是近几年崛起的一个免费软件 (Linux 下的 C shell 其实就是使用 tcsh 执行)，它虽然不是 UNIX 的标准配备，但是从许多地方您都可以下载到它。如果您是 C shell 的拥护者，笔者建议不妨试试 tcsh，因为您至少可以将它当作是 C shell 来使用。如果您愿意花点时间学习，您还可以享受许多它新增的优越功能，例如：

- 1) tcsh 提供了一个命令行 (command line) 编辑程序。
- 2) 提供了命令行补全功能。
- 3) 提供了拼写更正功能。它能够自动检测并且更正在命令行拼错的命令或是单字。
- 4) 危险命令侦测并提醒的功能，避免您一个不小心执行了 rm\* 这种杀伤力极大的命令。
- 5) 提供常用命令的快捷方式 (shortcut)。

bash 对 Bourne shell 是向下兼容 (backward compatible)，并融入许多 C shell 与 Korn shell 的功能。这些功能其实 C shell (当然也包括了 tcsh) 都有，只是过去 Bourne shell 都未支持。以下将介绍 bash 六点重要的改进：

1) 工作控制 (job control)。bash 支持了关于工作的讯号与指令，本章稍后会提及。

2) 别名功能 (aliases)。alias 命令是用来为一个命令建立另一个名称，它的运作就像一个宏，展开成为它所代表的命令。别名并不会替代掉命令的名称，它只是赋予那个命令另一个名字。

3) 命令历程 (command history)。

BASH shell 加入了 C shell 所提供的命令历程功能，它以 history 工具程序记录了最近您执行过的命令。命令是由 1 开始编号，默认最大值为 500。history 工具程序是一种短期记忆，记录您最近所执行的命令。要看看这些命令，您可以在命令行键入 history，如此将会显示最近执行过的命令的清单，并在前方加上编号。

这些命令在技术上每个都称为一个事件。事件描述的是一个已经采取的行动（已经被执行的命令）。事件是依照执行的顺序而编号，越近的事件其编号码越大，这些事件都是以它的编号或命令的开头字符来辨认的。history 工具程序让您参照一个先前发生过的事件，将它放在命令行上并允许您执行它。最简单的方法是用上下键一次放一个历程事件在您的命令列上；您并不需要先用 history 显示清单。按一次向上键会将最后一个历程事件放在您的命令列上，再按一次会放入上一个历史事件。按向下键则会将后一个事件放在命令行上。

4) 命令行编辑程序。

BASH shell 命令行编辑能力是内建的，让您在执行之前轻松地修改您输入的命令。若是您在输入命令时拼错了字，您不需重新输入整个命令，只需在执行命令之前使用编辑功能纠正错误即可。这尤其适合于使用冗长的路径名称当作参数的命令时。命令行编辑作业是 Emacs 编辑命令的一部分，您可以用 Ctrl+F 或向右键往后移一个字符，Ctrl+B 或向左键往回移一个字符。Ctrl+D 或 DEL 键会删除光标目前所在处的字符。要增加文字的话，您只需要将光标移到您要插入文字的地方并键入新字符即可。无论何时，您都可以按 ENTER 键执行命令。

5) 允许使用者自定义按键。

6) 提供更丰富的变量型态、命令与控制结构至 shell 中。

bash 与 tcsh 一样可以从许多网站上免费下载，它们的性质也十分类似，都是整合其前一代的产品然后增添新的功能，这些新增的功能主要着重在强化 shell 的程序设计能力以及让使用者能够自行定义自己偏好的作业环境。除了上述的五种 shell 之外，zsh 也是一个广为 UNIX 程序设计人员与进阶使用者所采用的 shell，zsh 基本上也是 Bourne shell 功能的扩充。

### 3. Shell 的使用

不论是哪一种 Shell，它最主要的功用都是解译使用者在命令行提示符号下输入的指令。Shell 语法分析命令行，把它分解成以空白区分开的符号 (token)，



在此空白包括了 Tab 键、空白和换行 (New Line)。如果这些字包含了 metacharacter, shell 将会评估 (evaluate) 它们的正确用法。另外, shell 还管理档案输入输出及幕后处理 (background processing)。在处理命令行之后, shell 会寻找命令并开始执行它们。

Shell 的另一个重要功用是提供个人化的使用者环境, 这通常在 shell 的初始化档案中完成 (.profile、.login、.cshrc、.tcshrc 等等)。这些档案包括了设定终端机键盘和定义窗口的特征, 设定变量, 定义搜寻路径、权限、提示符号和终端机类型, 以及设定特殊应用程序所需要的变量, 例如窗口、文字处理程序及程序语言的链接库。Korn shell 和 C shell 加强了个别化的能力: 增加例程、别名、和内建变量集以避免使用者误杀档案、不慎签出、并在当工作完成时通知使用者。

Shell 也能当解译性的程序语言 (interpreted programming language)。Shell 程序, 通常叫做命令文件, 它由列在档案内的命令所构成。此程序在编辑器中编辑 (虽然也可以直接在命令行下写程序, online scripting), 由 UNIX 命令和基本的程序结构, 例如变量的指定、测试条件和循环所构成。您不需要编译 shell 命令档。Shell 本身会解译命令档中的每一行, 就如同由键盘输入一样。shell 负责解译命令, 而使用者则必须了解这些命令能做什么。这本书的索引列出了一些有用的命令和它们的使用方法

#### 4. Shell 的功能

为了确保任何提示符号下输入的命令都能够适当地执行。shell 担任的工作包括有:

- 读取输入和语法分析命令列;
- 对特殊字符求值;
- 设立管线、转向、和幕后处理;
- 处理讯号;
- 设立程序来执行。

s6818 系列实验平台系统代码通过一个 shell 脚本 (/usr/local/src/s6818/x6818/mk)进行编译管理。文件参看平台上 mk 文件。

#### 六、实验步骤二：编译脚本分析

阅读理解编译脚本文件, 执行下面的步骤可以编译整个 Linux 系统:

- 在 Ubuntu 中单击菜单应用程序->附件->终端打开终端。
- 在终端中输入以下命令开始编译系统:

```
$ cd /usr/local/src/s6818/x6818
```

```
$ ./mk -c      # 清除源代码编译结果
```

```
$ ./mk -a # 编译所有源码: uboot, kernel, Android
```

编译成功后，将会在/usr/local/src/s6818/x6818/out/release 目录下看到编译生成的镜像文件 ubootpak.bin、boot.img、system.img，按照《产品手册》7.2 章节内容可以将编译好的镜像文件固化到实验平台中。

## 七、实验内容三：uboot 启动代码解析

简单地说，BootLoader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

Bootloader（引导加载程序）是系统加电后运行的第一段代码，一般运行的时间非常短，但是对于嵌入式系统来说，这段代码非常重要。在我们的台式电脑当中，引导加载程序由 BIOS（固件程序）和位于硬盘 MBR 中的操作系统引导加载程序（比如 NTLOADER，GRUB 和 LILO）一起组成。

在嵌入式系统当中没有像 BIOS 这样的固件程序，不过也有一些嵌入式 CPU 会在芯片内部嵌入一小段程序，一般用来将 bootloader 装进 RAM 中，有点类似 BIOS，但是功能比 BIOS 弱很多。在一般的典型系统中，整个系统的加载启动任务全由 bootloader 来完成。在 ARM 中，系统上电或复位时通常从地址 0x00000000 处开始执行，而在这个位置，通常安排的就是系统的 bootloader。通过这小段程序可以初始化硬件设备、建立内存空间映射图，从而将系统的软硬件环境设置到一个合适的状态！以为最终调用操作系统内核准备好正确的环境。

嵌入式 LINUX 系统从软件的角度可看成是 4 个层次：

- 1) 引导加载程序，包括固化在固件中（firmware）中的启动代码（可选）和 BOOTLOADER 两大部分。
- 2) 内核。特定于板子的定制内核以及控制内核引导系统的参数。
- 3) 文件系统。包括根文件系统和建立与 FLASH 内存设备上的文件系统。
- 4) 用户应用程序。特定于用户的应用程序，有时还包括一个 GUI。

bootloader 是严重地依赖于硬件而实现的，特别是在嵌入式世界。因此，在嵌入式世界里建立一个通用的 bootloader 几乎是不可能的。尽管如此，我们仍然可以对 bootloader 归纳出一些通用的概念来，以指导用户特定的 bootloader 设计与实现。bootloader 的启动流程：

大多数分为两个阶段，第一个阶段主要是包含依赖于 CPU 的体系结构的硬件初始化代码，通常都是用汇编语言来实现的。这个阶段的任务有：

- 基本的硬件设备初始化（屏蔽所有中断、关闭处理器内部指令/数据 CACHE 等）；
- 为第二阶段准备 RAM 空间；
- 如果是从某个固态存储媒质中，则复制 bootloader 的第二阶段代码到 RAM；
- 设置堆栈；

- 跳转到第二阶段的 C 程序入口点。

第二阶段通常是由 C 语言实现的，这个阶段的主要任务有：

- 初始化本阶段所要用到的硬件设备；
- 检测系统的内存映射；
- 将内核映像和根文件系统映像从 FLASH 读到 RAM；
- 为内核设置启动参数；
- 调用内核。

BOOTLOADER 调用 LINUX 内核的方法是直接跳转到内核的第一条指令处，即跳转 MEM\_START+0x8000 地址处，在跳转的时候必须满足下面的条件：

- CPU 寄存器：R0 为 0，R1 为机器类型 ID，R2 为启动参数，标记列表在 RAM 中的起始基地址；
- CPU 模式：必须禁止中断，CPU 设置为 SVC 模式；
- Cache 和 MMU 设置：MMU 必须关闭，指令 CACHE 可以打开也可以关闭，数据 CACHE 必须关闭。

U-Boot (Universal Bootloader) 是遵循 GPL 条款的开放源码项目。它是从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源码目录、编译形式与 Linux 内核很相似，事实上，不少 U-Boot 源码就是相应的 Linux 内核源程序的简化，尤其是一些设备的驱动程序，这从 U-Boot 源码的注释中能体现这一点。但是 U-Boot 不仅仅支持嵌入式 Linux 系统的引导，而且还支持 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 等嵌入式操作系统。其目前要支持的目标操作系统是 OpenBSD、NetBSD、FreeBSD、4.4BSD、Linux、SVR4、Esix、Solaris、Irix、SCO、Dell、NCR、VxWorks、LynxOS、pSOS、QNX、RTEMS、ARTOS。这是 U-Boot 中 Universal 的一层含义，另外一层含义则是 U-Boot 除了支持 PowerPC 系列的处理器外，还能支持 MIPS、x86、ARM、NIOS、XScale 等诸多常用系列的处理器。这两个特点正是 U-Boot 项目的开发目标，即支持尽可能多的嵌入式处理器和嵌入式操作系统。就目前为止，U-Boot 对 PowerPC 系列处理器支持最为丰富，对 Linux 的支持最完善。

U-Boot 提供两种操作模式：启动加载 (Boot loading) 模式和下载 (Downloading) 模式，并具有大型 Boot Loader 的全部功能。主要特性为：

- SCC/FEC 以太网支持；
- BOOTP/TFTP 引导；
- IP, MAC 预置功能；
- 在线读写 FLASH, DOC, IDE, IIC, EEROM, RTC；
- 支持串行口 kermi, S-record 下载代码；
- 识别二进制、ELF32、pImage 格式的 Image，对 Linux 引导有特别的支持；
- 监控(minitor)命令集：读写 I/O, 内存, 寄存器、内存、外设测试功能等；
- 脚本语言支持 (类似 BASH 脚本)；

- 支持 WatchDog, LCD logo, 状态指示功能等。

U-Boot 的功能是如此之强大, 涵盖了绝大部分处理器构架, 提供大量外设驱动, 支持多个文件系统, 附带调试、脚本、引导等工具, 特别支持 Linux, 为板级移植做了大量的工作。

## 八、实验步骤三: uboot 启动代码解析

### 1) 单独编译 u-boot

uboot 的默认配置文件为: x6818/uboot/include/configs/x6818.h, 如果要修改配置选项, 可以直接修改该文件, 然后在终端执行下面命令来编译 uboot 源码:

```
$ cd /usr/local/src/s6818/x6818
```

```
$ ./mk -u # 编译 uboot
```

编译成功后, 将会在/usr/local/src/s6818/x6818/out/release 目录下看到编译生成的镜像文件 ubootpak.bin, 按照《产品手册》7.2 章节内容可以将编译好的 uboot 镜像文件固化到实验平台中。

*说明: 用户也可以采用 uboot 的官方步骤进行编译, 如下:*

```
$ cd /usr/local/src/s6818/x6818/uboot
```

```
$ make distclean CROSS_COMPILE=../prebuilts/gcc/linux-x86/arm/arm-eabi-
```

```
4.7/bin/arm-eabi- # 清除之前的编译信息
```

```
$ make x6818_config CROSS_COMPILE=../prebuilts/gcc/linux-x86/arm/arm-eabi-
```

```
4.7/bin/arm-eabi- #配置 uboot编译参数
```

```
$ make ARCH=arm CROSS_COMPILE=../prebuilts/gcc/linux-x86/arm/arm-eabi-
```

```
4.7/bin/arm-eabi- #编译 uboot
```

### 2) u-boot 的命令使用

正确连接交叉串口线 (COM0), 在终端输入 minicom, 运行串口终端, 长按主板 Power 按键给实验平台上电, 在串口终端可以看到 uboot 的启动信息, 在启动 Linux 系统前快速按下电脑的空格键进入到 uboot 的命令行模式。

## 九、实验内容四: Linux 系统内核解析

内核简介:

内核, 是一个操作系统的核心。它负责管理系统的进程、内存、设备驱动程序、文件和网络系统, 决定着系统的性能和稳定性。

Linux 的一个重要的特点就是其源代码的公开性, 所有的内核源程序都可以在/usr/src/linux 下找到, 大部分应用软件也都是遵循 GPL 而设计的, 你都可以获取相应的源程序代码。

全世界任何一个软件工程师都可以将自己认为优秀的代码加入到其中, 由此

引发的一个明显的好处就是 Linux 修补漏洞的快速以及对最新软件技术的利用。而 Linux 的内核则是这些特点的最直接的代表。

想象一下，拥有了内核的源程序对你来说意味着什么？首先，我们可以了解系统是如何工作的。通过通读源代码，我们就可以了解系统的工作原理，这在 Windows 下简直是天方夜谭。其次，我们可以针对自己的情况，量体裁衣，定制适合自己的系统，这样就需要重新编译内核。

在 Windows 下是什么情况呢？相信很多人都被越来越庞大的 Windows 整得莫名其妙过。再次，我们可以对内核进行修改，以符合自己的需要。这意味着什么？没错，相当于自己开发了一个操作系统，但是大部分的工作已经做好了，你所要做的就是增加并实现自己需要的功能。在 Windows 下，除非你是微软的核心技术人员，否则就不用痴心妄想了。

内核版本号：

由于 Linux 的源程序是完全公开的，任何人只要遵循 GPL，就可以对内核加以修改并发布给他人使用。Linux 的开发采用的是集市模型（bazaar，与 cathedral--教堂模型--对应），为了确保这些无序的开发过程能够有序地进行，Linux 采用了双树系统。

一个树是稳定树（stable tree），另一个树是非稳定树（unstable tree）或者开发树（development tree）。一些新特性、实验性改进等都将首先在开发树中进行。如果在开发树中所做的改进也可以应用于稳定树，那么在开发树中经过测试以后，在稳定树中将进行相同的改进。一旦开发树经过了足够的发展，开发树就会成为新的稳定树。

开发树就体现在源程序的版本号中；源程序版本号的形式为 x.y.z：对于稳定树来说，y 是偶数；对于开发树来说，y 比相应的稳定树大一（因此，是奇数）。到目前为止，稳定树的最高版本是 2.2.16，最新发布的 RedHat7.0 所采用的就是 2.2.16 的内核；开发树的最新版本是 2.3.99。也许你已经发现和多网站上都有 2.4.0-test9-pre7 之类的内核，但是这并不是正式版本。内核版本的更新可以访问 <http://www.kernel.org>。

为什么重新编译内核？

Linux 作为一个自由软件，在广大爱好者的支持下，内核版本不断更新。新的内核修订了旧内核的 bug，并增加了许多新的特性。如果用户想要使用这些新特性，或想根据自己的系统度身定制一个更高效，更稳定的内核，就需要重新编译内核。

通常，更新的内核会支持更多的硬件，具备更好的进程管理能力，运行速度更快、更稳定，并且一般会修复老版本中发现的许多漏洞等，经常性地选择升级更新的系统内核是 Linux 使用者的必要操作内容。

为了正确的合理地设置内核编译配置选项，从而只编译系统需要的功能的代码，一般主要有下面四个考虑：

1. 自己定制编译的内核运行更快（具有更少的代码）；

2. 系统将拥有更多的内存（内核部分将不会被交换到虚拟内存中）；
3. 不需要的功能编译进入内核可能会增加被系统攻击者利用的漏洞；
4. 将某种功能编译为模块方式会比编译到内核内的方式速度要慢一些。

内核是 Linux 操作系统的核心，它管理所有的系统线程、进程、资源和资源分配。与其它操作系统不同的是，Linux 操作系统允许用户对内核进行重新设置。用户可以对内核进行“瘦身”，增加或消除对某些特定设备或子系统的支持。在开发嵌入式系统时，开发人员经常会减少系统对一些无用设备的支持，将节省下来的内存分配给各种应用软件。

Linux 内核对各种硬件和端口的支持要靠各种硬件驱动程序来实现。这些驱动程序可以被直接写入内核，也可以针对某些特定硬件在需要时自动加载。通常情况下，可以被自动加载进内核的内核编码称为自动加载内核模块。

Linux 内核的设置是通过内核设置编辑器完成的。内核设置编辑器可对每个内核设置变量进行描述，帮助用户决定哪些变量需要被清除，哪些需要写入内核，或者编成一个可加载内核模块在需要时进行加载。

内核是为众多应用程序提供对计算机硬件的安全访问的一部分软件，这种访问是有限的，并且内核决定一个程序在什么时候对某部分硬件操作多长时间。直接对硬件操作是非常复杂的，所以内核通常提供一种硬件抽象的方法来完成这些操作。硬件抽象隐藏了复杂性，为应用软件和硬件提供了一套简洁，统一的接口，使程序设计更为简单。

用户可以根据自己的需要编译内核。

## 十、实验步骤四：Linux 系统内核解析

Linux 内核的默认配置文件为：x6818/kernel/arch/arm/configs/x6818 defconfig，如果要修改配置选项，可以直接修改该文件，然后在终端执行下面命令来编译 Linux 内核源码：

```
$ cd /usr/local/src/s6818/x6818
```

```
$ ./mk -k # 编译 Linux 内核
```

也可以进入到图形配置界面对内核进行配置，然后编译 Linux 内核源码：

```
$ cd /usr/local/src/s6818/x6818/kernel
```

```
$ export PATH=../uboot/tools:$PATH
```

```
$ cp arch/arm/configs/x6818_defconfig .config
```

```
$ make menuconfig # 此时可以在该图形窗口进行选项的配置，保存后退出
```

```
$ make uImage ARCH=arm CROSS_COMPILE=../prebuilts/gcc/linux-x86/arm/arm-eabi-4.7/bin/arm-eabi- # 编译Linux内核
```

编译成功后，将会在/usr/local/src/s6818/kernel/arch/arm/boot/目录下看到编译生成的镜像文件 uImage，uImage 镜像文件需要根据 11.5 或者 11.6 内容打包

到文件系统 boot.img 中然后再固化到实验平台中。

## 十一、实验内容五：构建基本文件系统

busybox 是一个集成了一百多个最常用 linux 命令和工具的软件，他甚至还集成了一个 http 服务器和一个 telnet 服务器，而所有这一切功能却只有区区 1M 左右的大小。我们平时用的那些 linux 命令就好比是分立式的电子元件，而 busybox 就好比是一个集成电路，把常用的工具和命令集成压缩在一个可执行文件里，功能基本不变，而大小却小很多倍，在嵌入式 linux 应用中，busybox 有非常广的应用。另外，大多数 linux 发行版的安装程序中都有 busybox 的身影，安装 linux 的时候按 ctrl+alt+F2 就能得到一个控制台，而这个控制台中的所有命令都是指向 busybox 的链接。busybox 的小身材大作用的特性，给制作一张软盘的 linux 带来了及大方便。

busybox 是标准 Linux 工具的一个单个可执行实现。busybox 包含了一些简单的工具，例如 cat 和 echo，还包含了一些更大、更复杂的工具，例如 grep、find、mount 以及 telnet。有些人将 busybox 称为 Linux 工具里的瑞士军刀。简单的说 busybox 就好像是个大工具箱，它集成压缩了

Linux 的许多工具和命令，用户可以根据自己的需要定制一个 busybox。

Linux 根文件系统包含了除内核以外的所有 linux 系统在引导和管理时需要的工具，做为启动引导驱动，包含如下目录：bin, dev, etc, home, lib, mnt, proc, sbin, usr, var。还需要有一些基本的工具：sh, ls, cp, mv.....（位于/bin 目录中）；必要的配置文件：inittab, rc, fstab.....位于（/etc 目录中）；必要的设备文件：/dev/tty\*, /dev/console, /dev/men.....（位于/dev 目录中）； sh, ls 等工具必要的运行库：glibc。

## 十二、实验步骤五：构建基本文件系统

### 1. 编译 busybox:

在终端执行下面命令来编译 busybox:

1) 设置工作环境:

```
$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
```

```
$ mkdir -p /usr/local/src/s6818/project
```

2) 部署实验源码，将光盘：05-实验例程/第 11 章/11.6-busybox 文件夹拷贝到/usr/local/src/s6 818/project 路径下；

3) 执行下面的步骤来进行编译:

```
$ cd /usr/local/src/s6818/project/11.6- busybox $ tar xvf busybox-1.24.1.tar.gz
```

```
$ cd busybox-1.24.1
$ cp s6818_config .config
$ make
$ make CONFIG_PREFIX=./root-mini install
```

编译成功将生成一个 root-mini 文件夹，这样基于 busybox 所建立的最小文件系统目录树就做好了。

## 2. 制作最小根文件系统镜像：

在终端执行下面命令来编译 busybox：

```
$ cd /usr/local/src/s6818/project/11.6-busybox
$ cd root-mini
$ mkdir -p dev etc/init.d home/app mnt opt proc lib sys tmp var usr/lib media/sd0
media/sd1 media/usb0 media/usb1 lib/modules/3.4.39/firmware
$ cp -r /usr/local/src/s6818/arm-2009q3/arm-none-linux-gnueabi/libc/lib/* ./lib/
$ cp -r /usr/local/src/s6818/arm-2009q3/arm-none-linux-
gnueabi/libc/usr/lib/*.* ./usr/lib/
$ cp -r ../etc/* ./etc/
$ cp ../rz ./bin/
$ cp -r ../lib/* ./lib
$ cp ../uImage .
$ cd ..
$ chmod -R 777 root-mini
$ chmod +x mkfs
$ sudo ./mkfs
```

编译成功将生成一个 rootfs.img 系统镜像文件（rootfs.img 文件包含编译好的内核文件 uImage 和 busybox 制作的根文件系统，所以本实验要求必须先完成 11.4 章节实验生成 uImage 文件，并拷贝到本实验目录内）。

## 3. 固化最小根文件系统镜像：

参考《产品手册》7.2 章节固化新镜像文件，涉及到的修改步骤如下：

1) 打开 Zflasher for 6818 工具，在左边条目中选择“Linux”；

注意：选择要烧写的镜像文件一定要放在英文目录下（如：D:\images\rootfs.img）。





选择并勾选要烧写的镜像文件，点击开始刷机后，就可以更新系统镜像（ubootpak.bin、rootfs.img）。

固化成功后，重启实验平台，在串口超级终端可以看到 Linux 的启动打印消息。

## 实验二 LED 与按键驱动实验

### 一、实验目的

- 掌握嵌入式 GPIO 驱动程序的编写，工作原理。
- 熟悉 linux 环境下 LED 驱动程序的编写，运行。
- 熟悉 linux 环境下按键驱动程序的编写，测试及运行。

### 二、实验环境

- 硬件：电脑（推荐：主频 2GHz+，内存：1GB+）s6818 系列实验平台；
- 软件：ZEmberOS 操作系统。

### 三、实验内容一：LED 驱动实验

#### 1. LED 硬件原理图

Linux 下的设备驱动程序即是对具体硬件进行操作的程序，是操作系统内核和机器硬件之间的接口。设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作。

如下图所示，为 s6818 开发平台的 LED 部分硬件原理图如下：

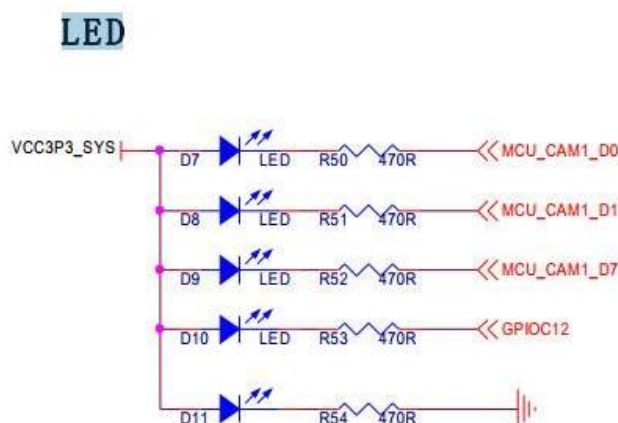


图 2-1 LED 部分原理图

s6818 底板上设计有 4 个用户可编程的 LED (D7~D10)，本驱动实验中我们四个灯一起使用。D7—D10 的正极分别与“+3.3V”相连， 负极分别间接与

s6818 处理器的 MCU\_CAM1\_D0 、MCU\_CAM1\_D1、MCU\_CAM1\_D7、GPIOC12 相连，即 MCU\_CAM1\_D0、MCU\_CAM1\_D1、MCU\_CAM1\_D7、GPIOC12 输出低电平时，LED 亮；输出高电平时，LED 灭。

## 2. Linux 系统下 LED 驱动的编写

设备驱动程序首先应包含设备的注册和释放函数，即 `static int init EXYNOS6818_led_init(void)` 和 `static void exit EXYNOS6818_led_exit(void)`。

/\*模块的初始化\*/

```
static int    init EXYNOS6818_led_init(void)
{
    int ret;

    ret = EXYNOS6818_led_ctrl_init(); if(ret)
    {
        printk(KERN_ERR "Apply: EXYNOS6818_LED_init--Fail !!!\n");
        return ret;
    }
    return 0;
}
```

/\*模块的退出\*/

```
static void    exit EXYNOS6818_led_exit(void)
{
    device_destroy(led_dev_class,DEVICE_NODE);
    class_destroy(led_dev_class);
    cdev_del(cdev_p);
    unregister_chrdev_region(num_dev,1);
}
```

EXYNOS6818\_led\_init 函数调用 EXYNOS6818\_led\_ctrl\_init()函数来注册设备的驱动程序。

```

/*LED 灯的初始化和 LED 设备驱动的加载*/static int EXYNOS6818_led_ctrl_init(void)
{
    int err, ret;
    struct device* temp=NULL;
    unsigned int gpio;

    gpio_free(PAD_GPIO_B+0);
    gpio_free(PAD_GPIO_A+30);
    gpio_free(PAD_GPIO_B+10);
    gpio_free(PAD_GPIO_C+12);
    ret = gpio_request(PAD_GPIO_B+0, "GPB0");
    if(ret)
        printk("mach-s6818: request gpio GPB0 fail");
    ret = gpio_request(PAD_GPIO_A+30, "GPB30");
    if(ret)
        printk("mach-s6818: request gpio GPB30 fail");
    ret = gpio_request(PAD_GPIO_B+10, "GPB10");
    if(ret)
        printk("mach-s6818: request gpio GPB10 fail");
    ret = gpio_request(PAD_GPIO_C+12, "GPC12");
    if(ret)
        printk("mach-s6818: request gpio GPC12 fail");

    gpio_direction_output(PAD_GPIO_A+30, 1);
    gpio_direction_output(PAD_GPIO_B+0, 1);
    gpio_direction_output(PAD_GPIO_B+10, 1);
    gpio_direction_output(PAD_GPIO_C+12, 1);

    /*动态注册 led_test 设备,num_dev 为动态分配出来的设备号(主设备号+次设备号)*/
    err=alloc_chrdev_region(&num_dev,0,1,DEVICE_LIST);
    if (err < 0) {
        printk(KERN_ERR "LED: unable to get device name %d/n", err);
        return err;
    }
}

```

```

/*动态分配 cdev 内存空间*/ cdev_p = cdev_alloc();
cdev_p->ops = &EXYNOS6818_led_ctrl_ops;


/*加载设备驱动*/
err=cdev_add(cdev_p,num_dev,1);
if(err){
    printk(KERN_ERR "LED: unable to add the device %d/n", err);
    return err;
}


/*在/sys/class 下创建 led_test 目录*/
led_dev_class=class_create(THIS_MODULE,DEVICE_LIST);
if(IS_ERR(led_dev_class))
{
    err=PTR_ERR(led_dev_class); goto unregister_cdev;
}


/* 基于 /sys/class/led_test 和 /dev 下面 创建 led_light 设备文件 */
temp=device_create(led_dev_class, NULL,num_dev, NULL, DEVICE_NODE);
if(IS_ERR(temp))
{
    err=PTR_ERR(temp);
    goto unregister_class;
}


return 0;


unregister_class:
    class_destroy(led_dev_class);
unregister_cdev:
    cdev_del(cdev_p);
    return err;
}

```

`alloc_chrdev_region` 函数是动态分配设备号；  
`cdev_alloc()`函数是动态分配内存空间；  
`cdev_add()`函数为动态加载驱动；  
`class_create` 函数是在`/sys/class` 下创建设备类型；  
`device_create` 函数是基于类的基础上创建设备文件。

`static void` `exit leds_ctrl_exit(void)`函数是在我们使用完此设备后，释放设备号、设备结构体内存。

`file_operations` 结构体是一个函数指针集合，定义能够在设备上进行的操作，比如 `open()`、`read()`、`write()`、`release()`、`ioctl()` 操作。驱动程序中定义的 `file_operations` 结构体如下：

```
/*定义具体的文件操作*/
static const struct file_operations EXYNOS6818_led_ctrl_ops={
.owner   = THIS_MODULE,
.open    = EXYNOS6818_led_open,
.read    =EXYNOS6818_led_read,
.write   =EXYNOS6818_led_write,
.release =EXYNOS6818_led_release,
};
```

变量 `open` 定义的 `EXYNOS6818_led_open` 用于打开 LED 设备，变量 `release` 定义的 `EXYNOS6818_led_release` 用于释放 LED 设备。变量 `read` 定义的 `EXYNOS6818_led_read` 则是读取灯的状态，变量 `write` 用于写控制 LED 灯的亮灭，其具体定义如下：

```
/*读取 LED 灯的状态*/
static ssize_t EXYNOS6818_led_read(struct file * file,char * buf,size_t count,loff_t * f_ops)
{
    /*从用户空间读取数据,获取 LED 灯的状态*/
    copy_to_user(buf, (char *)&led_status, sizeof(unsigned char));
    return sizeof(unsigned char);
}

/*定义实现 LED 灯的写操作*/
static ssize_t EXYNOS6818_led_write (struct file * file,const char * buf, size_t count,loff_t * f_ops)
{
    unsigned char status;
    if(count==1)
```

```

{
    /*向用户空间写数据,如果写失败, 则返回错误*/
    if(copy_from_user(&status, buf,sizeof(unsigned char)))
        return -EFAULT;
    set_led_status(status);
    return sizeof(unsigned char);
}else
    return -EFAULT;
}

```

其中 copy\_to\_user 和 copy\_from\_user 函数的具体含义如下:

unsigned long

copy\_to\_user(void user \*to, const void \*from, unsigned long n)

功能: 从内核空间拷贝一块儿数据到用户空间,只能用于用户空间;

To: 目标地址, 这个地址是用户空间的地址;

From: 源地址, 这个地址是内核空间的地址;

N: 将要拷贝的数据的字节数

返回值: 如果数据拷贝成功, 则返回零; 否则, 返回没有拷贝成功的数据字节数

unsigned long

copy\_from\_user(void \*to, const void user \*from, unsigned long n)

功能: 从用户空间拷贝数据到内核空间

To: 目标地址, 这个地址是内核空间的地址;

From: 源地址, 这个地址是用户空间的地址;

返回者: 如果数据拷贝成功, 则返回零; 否则, 返回没有拷贝成功的数据字节数

关于 LED 灯状态的控制代码如下:

```

/*设置 LED 灯的状态*/
static void set_led_status(unsigned char status)
{
    /*表示 LED 灯的状态是否发生变化*/
    unsigned char led_status_changed;

    led_status_changed= led_status^(status & 0xF);
}

```

```

/*数据变化检测*/ led_status=(status & 0xF);

/*如果 4 个 LED 灯的状态发生了变化*/
if(led_status_changed!=0x00)
{
    /*判断是否改变 LED1 灯的状态*/ if(led_status_changed&LED1)
    {
        if(led_status&LED1)
            gpio_set_value(PAD_GPIO_A+30, 0);
        else
            gpio_set_value(PAD_GPIO_A+30, 1);
    }

    /*判断是否改变 LED2 灯的状态*/
    if(led_status_changed&LED2)
    {
        if(led_status&LED2)
            gpio_set_value(PAD_GPIO_B+0, 0);
        else
            gpio_set_value(PAD_GPIO_B+0, 1);
    }

    /*判断是否改变 LED3 灯的状态*/
    if(led_status_changed&LED3)
    {
        if(led_status&LED3)
            gpio_set_value(PAD_GPIO_B+10, 0);
        else
            gpio_set_value(PAD_GPIO_B+10, 1);
    }

    /*判断是否改变 LED4 灯的状态*/
    if(led_status_changed&LED4)
    {
        if(led_status&LED4)

```



```

        gpio_set_value(PAD_GPIO_C+12, 0);
    else
        gpio_set_value(PAD_GPIO_C+12, 1);
    }

}

}

```

#### 四、实验步骤一：LED 驱动实验

##### 1. 编译：

###### 1) 设置工作环境：

```

$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
$ mkdir -p /usr/local/src/s6818/project

```

2) 部署实验源码，将光盘：05- 实验例程/ 第 13 章/13.2-led\_driver 文件夹拷贝到/usr/local/src/s6818/project 路径下；

###### 3) 编译并拷贝 led\_driver 程序：

```

$ cd /usr/local/src/s6818/project/13.2-led_driver/led_test
$ make
$ make install
$ make clean
$ cd /usr/local/src/s6818/project/13.2-led_driver/led_ctrl
$ make
$ make install
$ make clean

```

可执行文件会自动拷贝到/opt/tftp 目录下。

##### 2. 运行：

###### 1) 正确设置 ubuntu 系统的网络，保证网络通信正常

2) 准备好 s6818 实验平台，确保已经按照第 11 章节固化好 Linux 操作系统。

*附注：确保交叉网线和交叉串口线已经连接好主机和实验平台，在终端上用“ifconfig eth0 192.168.0.xxx ”命令设置 ip 地址。核对主机网卡的 ip 地址和 s6818 实验平台的 ip 地址为同一个网段。本实验测试时，主机网卡的 ip 地址为 192.168.0.205，s6818 实验平台的 ip 地址为 192.168.0.101。*

3) 运行 minicom 串口终端, 给实验平台加电, 进入 Linux 系统, 可以看到串口终端的启动打印信息。

4) 系统启动完成后, 在 minicom 下执行以下命令将 led\_driver 程序下载到 s6818 实验平台。

```
# tftp -g 192.168.0.205 -r./led_test -l/home/app/led_test
# tftp -g 192.168.0.205 -r./led.ko -l/lib/modules/3.4.39/led.ko
```

5) 给 led\_test 添加可执行权限:

```
# cd /home/app/
# chmod 777 led_test
```

6) 加载 led 驱动:

```
# insmod /lib/modules/3.4.39/led.ko
```

7) 运行 led\_test:

```
# ./led_test
led status:1
led status:2
led status:4
led status:8
led status:1
led status:2
led status:4
.....
```

可以观察到 s6818 开发平台上的 D7~D10 循环依次被点亮。

## 五、实验内容二：按键驱动设计实验

### 1. 按键电路原理

按键同样使用 GPIO 接口, 但按键本身需要外部的输入, 按键硬件驱动原理图如下图所示。在下图的 1×6 矩阵按键 (SW1~SW6) 电路中, 使用 7 个输入 (MCU\_KEY\_VOLUP、MCU\_KEY\_VOLDN、GPIOB9、GPIOA28、GPIOB8、GPIOC7 和 MCU\_CAM1\_D2)。

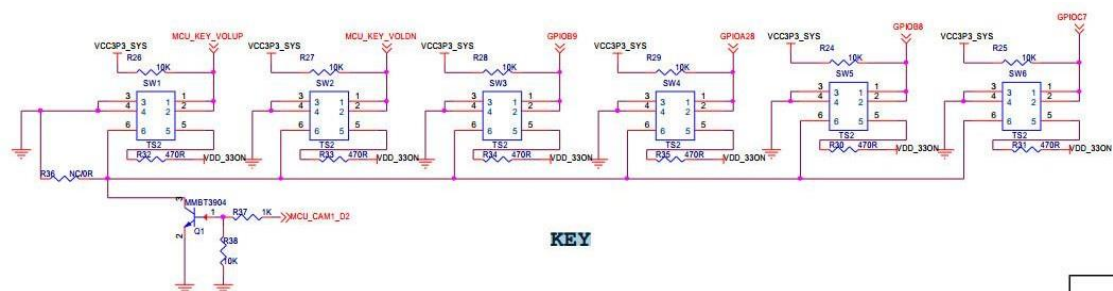


图2-2 按键电路

按键入口对应的核心板接口电路如下图所示：

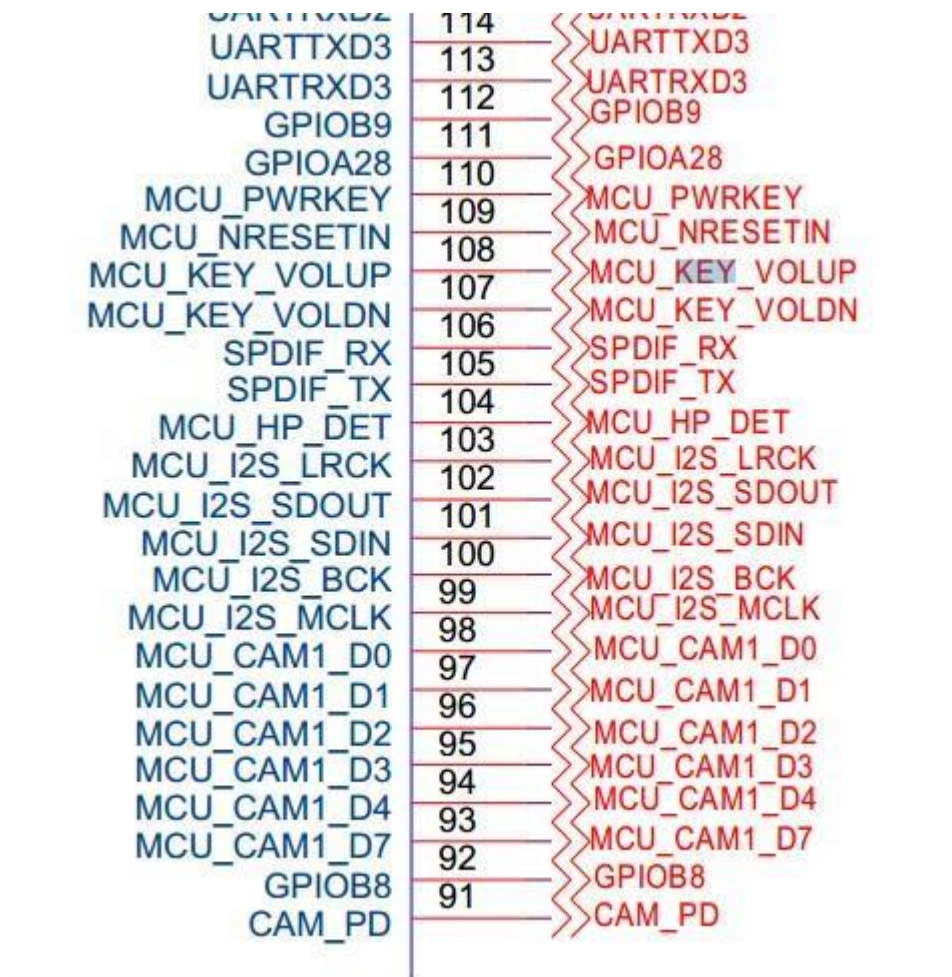


图 2-3 核心板接口电路

当其中一个 SW 按键被按下，通过查询方式就可以检测到是哪一个接口有输入信号，从而控制相应的操作。

以上的讨论都是在按键的理想状态下进行的，但实际的按键动作会在短时间（几毫秒至几十毫秒）内产生信号抖动。例如，当按键被按下时，其动作就像弹簧的若干次往复运动，将产生几个脉冲信号。一次按键操作将会产生若干次按键中断，从而会产生抖动现象。因此驱动程序中必须要解决去除抖动所产生的毛刺信号的问题。

## 2. Linux 系统下按键驱动的编写

Linux 内核已经对 gpio 接口的按键做了支持，具体驱动文件参考 kernel/drivers/input/keyboard/nxp\_io\_key.c

在编译内核的时候要增加对 gpio\_keys 的支持。可以通过 make menuconfig 来选择

```
Device Drivers --->
Input device support --->
```

```

[*]   Keyboards      --->
<*>   SLsiAP push Keypad support

```

然后在文件 `kernel/arch/arm/plat-s5p6818/drone/device.c` 中增加按键的平台资源，具体定义如下

```

static unsigned int button_gpio[] = CFG_KEYPAD_KEY_BUTTON; static unsigned int
button_code[] = CFG_KEYPAD_KEY_CODE;

```

```

struct nxp_key_plat_data key_plat_data = {
    .bt_count = ARRAY_SIZE(button_gpio),
    .bt_io     = button_gpio,
    .bt_code   = button_code,
    .bt_repeat = CFG_KEYPAD_REPEAT,
};

```

```

static struct platform_device key_plat_device = {
    .name     = DEV_NAME_KEYPAD,
    .id      = -1,
    .dev      = {
        .platform_data = &key_plat_data
    },
};

```

`kernel/drivers/input/keyboard/nxp_io_key.c` 关键代码分析。

```

static int      init nxp_key_init(void)
{
    return platform_driver_register(&key_plat_driver);    //注册设备驱动。
}

```

设备驱动注册之后会调用 `nxp_key_probe()` 去查找具体的按键设备。

```

static int nxp_key_probe(struct platform_device *pdev)
{
    //此处的 nxp_key_plat_data 即之前在 device.c 文件定义的按键资源 struct
    nxp_key_plat_data * plat = pdev->dev.platform_data;

    struct key_info *key = NULL;

```

```
struct key_code *code = NULL;
struct input_dev *input = NULL;
int i, keys;
int ret = 0;
```

```
pr_debug("%s (device name:%s, id:%d)\n",      func , pdev->name, pdev->id);
```

```
key = kzalloc(sizeof(struct key_info), GFP_KERNEL);
if (! key) {
    pr_err("fail, %s allocate driver info ...\n", pdev->name);
    return -ENOMEM;
}
```

```
keys = plat->bt_count;
code = kzalloc(sizeof(struct key_code)*keys, GFP_KERNEL);
if (NULL == code) {
    pr_err("fail, %s allocate key code ...\n", pdev->name);
    ret = -ENOMEM;
    goto err_mm;
}
```

```
input = input_allocate_device(); //从内核分配一个输入设备
if (NULL == input) {
    pr_err("fail, %s allocate input device\n", pdev->name);
    ret = -ENOMEM;
    goto err_mm;
}
```

```
key->input = input;
key->keys = keys;
key->code = code;
key->resume_delay_ms = plat->resume_delay_ms;
key_input = key->input;
```

```
INIT_DELAYED_WORK(&key->resume_work, nxp_key_resume_work);
```

```
input->name = "Nexell Keypad";  
input->phys = "nexell/input0";  
input->id.bustype = BUS_HOST;  
input->id.vendor = 0x0001;  
input->id.product = 0x0002;  
input->id.version = 0x0100;  
input->dev.parent = &pdev->dev;  
input->keycode = plat->bt_code;  
input->keycodesize = sizeof(plat->bt_code[0]);  
input->keycodemax = plat->bt_count * 2; // for long key  
input->evbit[0] = BIT_MASK(EV_KEY);
```

```
if (plat->bt_repeat)  
    input->evbit[0] |= BIT_MASK(EV_REP);
```

```
input_set_capability(input, EV_MSC, MSC_SCAN);  
input_set_drvdata(input, key);
```

```
ret = input_register_device(input); //向内核注册按键输入设备  
if (ret) {  
    pr_err("fail, %s register for input device ...\n", pdev->name);  
    goto err_mm;  
}
```

```
for (i=0; keys > i; i++, code++) {  
    code->io = plat->bt_io[i];  
    code->keycode = plat->bt_code[i];  
    code->detect_high = plat->bt_detect_high ? plat->bt_detect_high[i]: 0;  
    code->val = i;  
    code->info = key;  
    code->keystate = KEY_STAT_RELEASE;
```

```
code->kcode_wq = create_singlethread_workqueue(pdev->name);
```

```

if (!code->kcode_wq) {
    ret = -ESRCH;
    goto err_irq;
}

ret = request_irq(gpio_to_irq(code->io), nxp_key_irqhnd,
(IRQF_SHARED | IRQ_TYPE_EDGE_BOTH), pdev->name, code);
if (ret) {
    pr_err("fail, gpio[%d] %s request irq...\n", code->io, pdev->name);
    goto err_irq;
}

set_bit(code->keycode, input->keybit);
INIT_DELAYED_WORK(&code->kcode_work, nxp_key_event_wq);
pr_debug("[%d] key io=%3d, code=%4d\n", i, code->io, code->keycode);
}

platform_set_drvdata(pdev, key);

return ret;

err_irq:
for (--i; i >= 0; i--) {
    cancel_work_sync(&code[i].kcode_work.work);
    destroy_workqueue(code[i].kcode_wq);
    free_irq(gpio_to_irq(code[i].io), &code[i]);
}
input_free_device(input);

err_mm:
if (code)
    kfree(code);

if (key)
    kfree(key);

```

```

        return ret;
    }

```

`gpio_keys_probe()` 函数主要初始化按键 `gpio` 并向内核注册相应的输入设备。然后通过 `nxp_key_irqhnd()` 处理按键中断。

```

static irqreturn_t nxp_key_irqhnd(int irqno, void *dev_id)
{
    struct key_code *code = dev_id;

    queue_delayed_work(code->kcode_wq,
        &code->kcode_work, DELAY_WORK_JIFFIES);

    return IRQ_HANDLED;
}

```

`gpio_keys_work_func()` 函数为工作线程，处理按键事件

```

static void gpio_keys_work_func(struct work_struct *work)
{
    struct gpio_button_data *bdata =
        container_of(work, struct gpio_button_data, work);

    gpio_keys_report_event(bdata); //向内核上报按键事件
}

```

由于按键驱动在内核文件中已经编译好，因此本实验只需要编写按键测试程序进行测试即可。

## 六、实验步骤二：按键驱动设计实验

### 1. 编译：

#### 1) 设置工作环境：

```
$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
```

```
$ mkdir -p /usr/local/src/s6818/project
```

#### 2) 部署实验源码，将光盘：05-实验例程/第 13 章/13.3-button\_test 文件夹



拷贝到/usr/local/src/s6818/project 路径下；

3) 编译并拷贝 button\_test 程序：

```
$ cd /usr/local/src/s6818/project/13.3-button_test
```

```
$ make
```

```
$ make install
```

```
$ make clean
```

可执行文件会自动拷贝到/opt/tftp 目录下。

## 2. 运行：

1) 正确设置ubuntu 系统的网络，保证网络通信正常

2) 准备好s6818 实验平台，确保已经按照第 11 章节固化好 Linux 操作系统。

*附注：确保交叉网线和交叉串口线已经连接好主机和实验平台，在终端上用“ifconfig eth0 192.168.0.xxx”命令设置 ip 地址。核对主机网卡的ip 地址和 s6818 实验平台的 ip 地址为同一个网段。本实验测试时，主机网卡的 ip 地址为 192.168.0.205，s6818 实验平台的 ip 地址为192.168.0.101。*

3) 运行 minicom 串口终端，给实验平台加电，进入 Linux 系统，可以看到串口终端的启动打印信息。

4) 系统启动完成后，在minicom 下执行以下命令将 button\_test 程序下载到 s6818 实验平台。

```
# tftp -g 192.168.0.205 -r./button_test -l/home/app/button_test
```

5) 给 button\_test 添加可执行权限

```
#cd /home/app/
```

```
# chmod 777 button_test
```

6) 运行 button\_test，按下实验平台上的按键即可以输出按键的键值代码信息：

```
root@s6818 /home/app]#./button_test
```

```
Event: time 811.272078, code 116 , value 1
```

```
Event: time 811.272086, Event: time 811.392118, code 116 , value 0
```

```
Event: time 811.392122, Event: time 811.397076, code 116 , value 1
```

```
Event: time 811.397082, Event: time 811.458069, code 116 , value 0
```

```
Event: time 811.458074, Event: time 812.704076, code 139 , value 1
```

```
Event: time 812.704083, Event: time 812.903068, code 139 , value 0
```

```
Event: time 812.903072, Event: time 813.819074, code 102 , value 1
```

```
Event: time 813.819080, Event: time 813.989071, code 102 , value 0
```

```
Event: time 813.989076, Event: time 814.961080, code 158 , value 1
```

```
Event: time 814.961088, Event: time 815.103073, code 158 , value 0
```

```
Event: time 815.103077, Event: time 815.821073, code 114 , value 1
```

Event: time 815.821080, Event: time 816.021067, code 114 , value 0  
Event: time 816.021071, Event: time 816.646073, code 115 , value 1  
Event: time 816.646079, Event: time 816.846054, code 115 , value 0  
Event: time 816.846059, Event: time 817.512061, code 212 , value 1  
Event: time 817.512067, Event: time 817.688071, code 212 , value 0

## 七、实验任务

本实验任务需要把LED实验和按键实验结合在一起，从按键实验中我们可以得到实验箱每个按键的键值，本任务要求用四个按键去控制实验箱的四个LED灯，要求：

- (1) 同一按键控制同一个LED灯，要求按一下灯亮，再按灯灭。
- (2) 相互之间互不干扰，一个按键开关不影响其它三个LED灯状态。

提示：需要用到C语言位运算编程技术。

## 实验三 LCD 与触摸屏实验

### 一、实验目的

- 掌握嵌入式 LCD 驱动程序的编写，实现。
- 了解 linux 环境下 LCD 显示屏的工作原理。
- 掌握嵌入式触摸屏驱动程序的编写，实现。
- 了解 linux 环境下触摸屏屏的工作原理。

### 二、实验环境

- 硬件：电脑（推荐：主频 2GHz+，内存：1GB+）s6818 系列实验平台；
- 软件：ZEmberOS 操作系统。

### 三、实验内容一：LCD 显示实验

#### 1. LCD 的工作原理

要使一块 LCD 正常的显示文字或图像，不仅需要 LCD 驱动器，而且还需要相应的 LCD 控制器。在通常情况下，生产厂商把 LCD 驱动器会以 COF/COG 的形式与 LCD 玻璃基板制作在一起，而 LCD 控制器则是由外部的电路来实现，现在很多的 MCU 内部都集成了 LCD 控制器，如 EXYNOS6818 等。通过 LCD 控制器就可以产生 LCD 驱动器所需要的控制信号来控制 STN/TFT 屏了。LCD 控制器可以通过编程支持不同 LCD 屏的要求，例如行和列像素数，数据总线宽度，接口时序和刷新频率等。

LCD 控制器的主要作用，是将定位在系统存储器中的显示缓冲区中的 LCD 图像数据传送到外部

LCD 驱动器，并产生必要的控制信号，例如 RGB\_VSYNC, RGB\_HSYNC, RGB\_VCLK 等。

如下图所示是 EXYNOS6818 的 LCD 控制器内部结构框图如下：

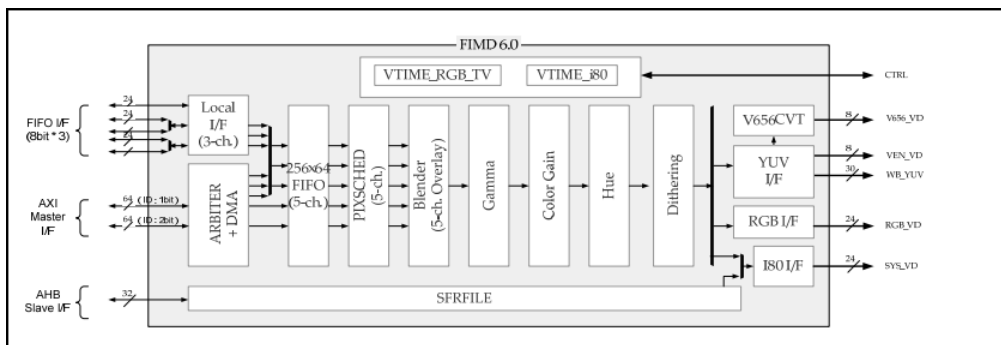


图 3-1 LCD 控制器结构框图

主要由 VSFR, VDMA, VPRCS, VTIME 和视频时钟产生器几个模块组成:

1) VSFR 由 121 个可编程控制器组, 一套 gamma LUT 寄存器组 (包括 64 个寄存器), 一套 i80 命令寄存器组 (包括 12 个寄存器) 和 5 块 256\*32 调色板存储器组成, 主要用于对 lcd 控制器进行配置。

2) VDMA 是 LCD 专用的 DMA 传输通道, 可以自动从系统总线上获取视频数据传送到 VPRCS, 无需 CPU 干涉。

3) VPRCS 收到数据后组成特定的格式 (如 16bpp 或 24bpp), 然后通过数据接口 (RGB\_VD, VEN\_VD, V656\_VD or SYS\_VD) 传送到外部 LCD 屏上。

4) VTIME 模块由可编程逻辑组成, 负责不同 lcd 驱动器的接口时序控制需求。VTIME 模块产生

RGB\_VSYNC, RGB\_HSYNC, RGB\_VCLK, RGB\_VDEN, VEN\_VSYNC 等信号。

主要特性:

- 1) 支持 4 种接口类型: RGB/i80/ITU 601(656)/YTU444
- 2) 支持单色、4 级灰度、16 级灰度、256 色的调色板显示模式
- 3) 支持 64K 和 16M 色非调色板显示模式
- 4) 支持多种规格和分辨率的 LCD
- 5) 虚拟屏幕最大可达 16MB
- 6) 5 个 256\*32 位调色板内存
- 7) 支持透明叠加

## 2. 外部接口电路

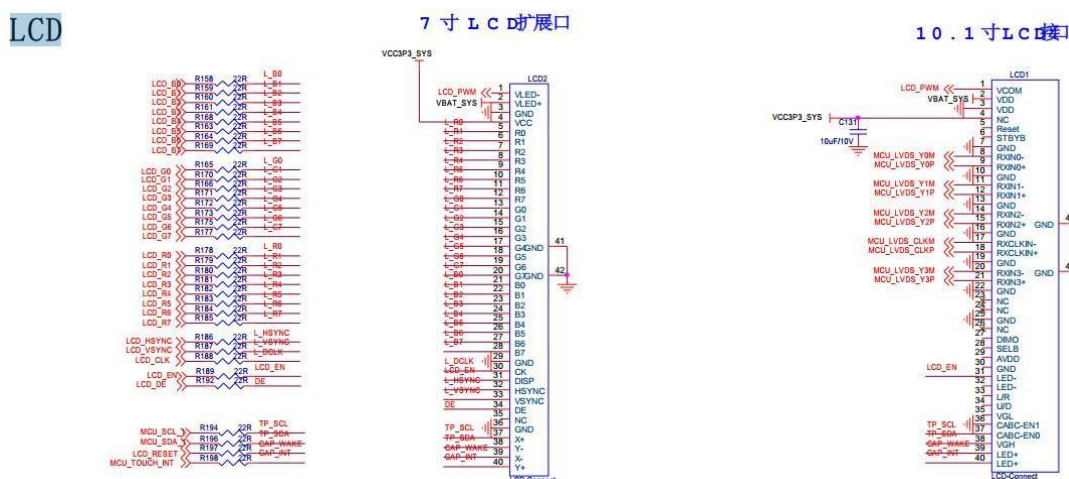


图 3-2 LCD 控制器接口电路

## 3. 数据结构及接口函数

从帧缓冲设备驱动程序结构看，该驱动主要跟 fb\_info 结构体有关，该结构体记录了帧缓冲设备的全部信息，包括设备的设置参数、状态以及对底层硬件操作的函数指针。在 Linux 中，每一个帧缓冲设备都必须对应一个 fb\_info，fb\_info 在/linux/fb.h 中的定义如下：(只列出重要的一些)

```
struct fb_info {
    int node;
    int flags;
    struct fb_var_screeninfo var; /*LCD 可变参数结构体*/
    struct fb_fix_screeninfo fix; /*LCD 固定参数结构体*/
    struct fb_monspecs monspecs; /*LCD 显示器标准*/
    struct work_struct queue;      /*帧缓冲事件队列*/
    struct fb_pixmap pixmap;       /*图像硬件 mapper*/
    struct fb_pixmap sprite; /*光标硬件 mapper*/
    struct fb_cmap cmap;          /*当前的颜色表*/
    struct fb_videomode *mode; /*当前的显示模式*/
#ifdef CONFIG_FB_BACKLIGHT
    struct backlight_device *bl_dev; /*对应的背光设备*/
    struct mutex bl_curve_mutex;
    u8 bl_curve[FB_BACKLIGHT_LEVELS]; /*背光调整*/
#endif
#ifdef CONFIG_FB_DEFERRED_IO
    struct delayed_work deferred_work; struct fb_deferred_io *fbdefio;
#endif
    struct fb_ops *fbops; /*对底层硬件操作的函数指针*/
    struct device *device;
    struct device *dev; /*fb 设备*/
    int class_flag;
#ifdef CONFIG_FB_TILEBLITTING
    struct fb_tile_ops *tileops; /*图块 Blitting*/
#endif
    char iomem *screen_base; /*虚拟基地址*/
    unsigned long screen_size; /*LCD IO 映射的虚拟内存大小*/
    void *pseudo_palette; /*伪 16 色颜色表*/
#define FBINFO_STATE_RUNNING 0
#define FBINFO_STATE_SUSPENDED 1
    u32 state; /*LCD 的挂起或恢复状态*/
    void *fbcon_par;
```

```
void *par;
};
```

其中，比较重要的成员有 `struct fb_var_screeninfo var`、`struct fb_fix_screeninfo fix` 和 `struct fb_ops *fbops`，他们也都是结构体。

`fb_var_screeninfo` 结构体主要记录用户可以修改的控制器的参数，比如屏幕的分辨率和每个像素的比特数等，该结构体定义如下：

```
struct fb_var_screeninfo {
    ____u32 xres;          /*可见屏幕一行有多少个像素点*/
    ____u32 yres;          /*可见屏幕一列有多少个像素点*/
    ____u32 xres_virtual;  /*虚拟屏幕一行有多少个像素点*/
    ____u32 yres_virtual;  /*虚拟屏幕一列有多少个像素点*/
    ____u32 xoffset;       /*虚拟到可见屏幕之间的行偏移*/
    ____u32 yoffset;       /*虚拟到可见屏幕之间的列偏移*/
    ____u32 bits_per_pixel; /*每个像素的位数即BPP*/

    u32 grayscale;        /*非0 时，指的是灰度*/

    struct fb_bitfield red; /*fb缓存的R位域*/
    struct fb_bitfield green; /*fb缓存的G位域*/
    struct fb_bitfield blue; /*fb缓存的B位域*/
    struct fb_bitfield transp; /*透明度*/
    ____u32 nonstd;        /* != 0 非标准像素格式*/
    ____u32 activate;
    ____u32 height;        /*高度*/
    ____u32 width;         /*宽度*/
    ____u32 accel_flags;
```

```

/*定时：除了pixclock本身外，其他的都以像素时钟为单位*/
____u32 pixclock;    /*像素时钟(皮秒)*/

____u32 left_margin; /*行切换，从同步到绘图之间的延迟*/
____u32 right_margin; /*行切换，从绘图到同步之间的延迟*/
____u32 upper_margin; /*帧切换，从同步到绘图之间的延迟*/
____u32 lower_margin; /*帧切换，从绘图到同步之间的延迟*/

____u32 hsync_len;    /*水平同步的长度*/
____u32 vsync_len;    /*垂直同步的长度*/

____u32 sync;

____u32 vmode;

____u32 rotate;

____u32 reserved[5]; /*保留*/

};

```

而 fb\_fix\_screeninfo 结构体又主要记录用户不可以修改的控制器的参数，比如屏幕缓冲区的物理地址和长度等，该结构体的定义如下：

```

struct fb_fix_screeninfo {

char id[16];          /*字符串形式的标示符 */

unsigned long smem_start; /*fb缓存的开始位置 */
____u32 smem_len;      /*fb缓存的长度 */
____u32 type;          /*看FB_TYPE_*/
____u32 type_aux;      /*分界*/
____u32 visual;        /*看FB_VISUAL_*/
____u16 xpanstep;      /*如果没有硬件panning就赋值为 0 */
____u16 ypanstep;      /*如果没有硬件panning就赋值为 0 */
____u16 ywrapstep;     /*如果没有硬件ywrap就赋值为 0 */
____u32 line_length;   /*一行的字节数 */

unsigned long mmio_start; /*内存映射IO的开始位置*/
____u32 mmio_len;      /*内存映射IO的长度*/
____u32 accel;

____u16 reserved[3]; /*保留*/

};

```

fb\_ops 结构体是对底层硬件操作的函数指针，该结构体中定义了对硬件的操作有:(这里只列出了常用的操作)

```
struct fb_ops {  
  
    struct module *owner;  
  
    //检查可变参数并进行设置  
  
    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);  
  
    //根据设置的值进行更新，使之有效  
  
    int (*fb_set_par)(struct fb_info *info);  
  
    //设置颜色寄存器  
  
    int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green, unsigned blue,  
                        unsigned transp, struct fb_info *info);  
  
    //显示空白  
  
    int (*fb_blank)(int blank, struct fb_info *info);  
  
    //矩形填充  
  
    void (*fb_fillrect) (struct fb_info *info, const struct fb_fillrect *rect);  
  
    //复制数据  
  
    void (*fb_copyarea) (struct fb_info *info, const struct fb_copyarea *region);  
  
    //图形填充  
  
    void (*fb_imageblit) (struct fb_info *info, const struct fb_image *image);  
  
};
```

### 3. LCD 参数配置

在 EXYNOS6818 中，LCD 控制器被集成在芯片的内部作为一个相对独立的单元，所以 Linux 把它看做是一个平台设备，故在内核代码 kernel/arch/arm/plat-s5p6818/drone 中定义有 LCD 相关的平台设备及资源，代码如下：



```

/* Default to lcd vs070cxn */

int CFG_DISP_PRI_SCREEN_LAYER = 0;

int CFG_DISP_PRI_SCREEN_RGB_FORMAT = MLC_RGBFMT_X8R8G8B8;

int CFG_DISP_PRI_SCREEN_PIXEL_BYTE = 4;

int CFG_DISP_PRI_SCREEN_COLOR_KEY = 0x090909;

int CFG_DISP_PRI_VIDEO_PRIORITY =

2;

int CFG_DISP_PRI_BACK_GROUND_COLOR = 0x000000;

int CFG_DISP_PRI_MLC_INTERLACE = 0;


int  CFG_DISP_PRI_LCD_WIDTH_MM = 152.4;

int  CFG_DISP_PRI_LCD_HEIGHT_MM = 91.44;

int

CFG_DISP_PRI_RESOL_WIDT

H = 1024;

int CFG_DISP_PRI_RESOL_HEIGHT = 600;

int CFG_DISP_PRI_HSYNC_SYNC_WIDTH = 20;

int CFG_DISP_PRI_HSYNC_BACK_PORCH = 140;

int CFG_DISP_PRI_HSYNC_FRONT_PORCH = 160;

int

CFG_DISP_PRI_HSYNC_ACTIVE_

HIGH = 1;

int CFG_DISP_PRI_VSYNC_SYNC_WIDTH = 3;

int CFG_DISP_PRI_VSYNC_BACK_PORCH = 20;

int CFG_DISP_PRI_VSYNC_FRONT_PORCH = 12;

int CFG_DISP_PRI_VSYNC_ACTIVE_HIGH = 1;

```

```

int CFG_DISP_PRI_CLKGEN0_SOURCE = DPC_VCLK_SRC_PLL2;int
CFG_DISP_PRI_CLKGEN0_DIV = 15;

int CFG_DISP_PRI_CLKGEN0_DELAY = 0;

int CFG_DISP_PRI_CLKGEN0_INVERT = 0;

int CFG_DISP_PRI_CLKGEN1_SOURCE = DPC_VCLK_SRC_VCLK2;

int CFG_DISP_PRI_CLKGEN1_DIV = 1;

int CFG_DISP_PRI_CLKGEN1_DELAY = 0;

int CFG_DISP_PRI_CLKGEN1_INVERT = 0;

int CFG_DISP_PRI_CLKSEL1_SELECT = 0;

int CFG_DISP_PRI_PADCLKSEL = DPC_PADCLKSEL_VCLK;

int CFG_DISP_PRI_PIXEL_CLOCK = (780000000 / 15);

int CFG_DISP_PRI_OUT_SWAPRB = 0;

int CFG_DISP_PRI_OUT_FORMAT = DPC_FORMAT_RGB888;

int CFG_DISP_PRI_OUT_YCORDER = DPC_YCORDER_CbYCrY;int

CFG_DISP_PRI_OUT_INTERLACE = 0;

int CFG_DISP_PRI_OUT_INVERT_FIELD = 0;

int CFG_DISP_LCD_MPY_TYPE = 0;
int CFG_DISP_LVDS_LCD_FORMAT = LVDS_LCDFORMAT_JEIDA; int
CFG_DISP_HDMI_USING = 0;

int CFG_DISP_MIPI_PLLPMS = 0x2281; int CFG_DISP_MIPI_BANDCTL = 0x7; int
CFG_DISP_MIPI_PLLCTL = 0;
int CFG_DISP_MIPI_DPHYCTL = 0;
static struct mipi_reg_val mipidef[] = {
{0, 0, 0, {0}},
};
struct mipi_reg_val * CFG_DISP_MIPI_INIT_DATA = &mipidef[0];

extern void x6818_lcd_select(void);
static unsigned char lcdname[32] = "vs070cxn"; static int init lcd_setup(char * str)
{
if((str != NULL) && (*str != '\0')) strcpy(lcdname, str);

```

```

x6818_lcd_select(); return 1;
}

    setup("lcd=", lcd_setup);
else if(strcmp(lcdname, "vs070cxn") == 0)
{
CFG_DISP_PRI_RESOL_WIDTH = 1024;
CFG_DISP_PRI_RESOL_HEIGHT = 600;


CFG_DISP_PRI_HSYNC_SYNC_WIDTH = 20;
CFG_DISP_PRI_HSYNC_BACK_PORCH = 140;
CFG_DISP_PRI_HSYNC_FRONT_PORCH = 160;
CFG_DISP_PRI_HSYNC_ACTIVE_HIGH = 1;
CFG_DISP_PRI_VSYNC_SYNC_WIDTH = 3;
CFG_DISP_PRI_VSYNC_BACK_PORCH = 20;
CFG_DISP_PRI_VSYNC_FRONT_PORCH = 12;
CFG_DISP_PRI_VSYNC_ACTIVE_HIGH = 1;


CFG_DISP_PRI_CLKGEN0_DIV = 15;
CFG_DISP_PRI_PIXEL_CLOCK = (780000000 / CFG_DISP_PRI_CLKGEN0_DIV);
}

```

除此之外，Linux 还在/arch/arm/mach-s5p6818/ 中为 LCD 平台设备定义了一个 s3c\_platform\_fb 结构体，该结构体主要是记录 LCD 的硬件参数信息(比如该结构体的 s3c\_fb\_lcdy 成员结构中就用于记录 LCD 的屏幕尺寸、屏幕信息、可变的屏幕参数、LCD 配置寄存器等)，这样在写驱动的时候就直接使用这个结构体。下面，我们来看一下内核是如果使用这个结构体的。在/arch/arm/mach-s5p6818/dev-display.c 中定义有：

```

/*
 * LCD platform device
 */

#if defined (CONFIG_NXP_DISPLAY_LCD)

static struct disp_vsync_info____lcd_vsync = {

    /* default parameters refer to cfg_main.h */

    #if defined(CFG_DISP_PRI_RESOL_WIDTH) &&
    defined(CFG_DISP_PRI_RESOL_HEIGHT)

```

```

.h_active_len = CFG_DISP_PRI_RESOL_WIDTH,
.h_sync_width = CFG_DISP_PRI_HSYNC_SYNC_WIDTH,
.h_back_porch = CFG_DISP_PRI_HSYNC_BACK_PORCH,
.h_front_porch = CFG_DISP_PRI_HSYNC_FRONT_PORCH,
.h_sync_invert = CFG_DISP_PRI_HSYNC_ACTIVE_HIGH,
.v_active_len = CFG_DISP_PRI_RESOL_HEIGHT,
.v_sync_width = CFG_DISP_PRI_VSYNC_SYNC_WIDTH,
.v_back_porch = CFG_DISP_PRI_VSYNC_BACK_PORCH,
.v_front_porch = CFG_DISP_PRI_VSYNC_FRONT_PORCH,
.v_sync_invert = CFG_DISP_PRI_VSYNC_ACTIVE_HIGH,
.pixel_clock_hz = CFG_DISP_PRI_PIXEL_CLOCK,
.clk_src_lv0 = CFG_DISP_PRI_CLKGEN0_SOURCE,
.clk_div_lv0 = CFG_DISP_PRI_CLKGEN0_DIV,
.clk_src_lv1 = CFG_DISP_PRI_CLKGEN1_SOURCE,
.clk_div_lv1 =
CFG_DISP_PRI_CLKGEN1_DIV,

#endif
};

static struct disp_lcd_param lcd_devpar;

static struct nxp_lcd_plat_data lcd_data = {
    .display_in = DISPLAY_INPUT(CONFIG_NXP_DISPLAY_LCD_IN),
    .display_dev = DISP_DEVICE_LCD,
    .vsync = &__lcd_vsync,
    .dev_param = (union disp_dev_param*)&__lcd_devpar,
};

static struct platform_device lcd_device = {
    .name = DEV_NAME_LCD,
    .id = -1,
    .dev = {

```

```

        .platform_data = &lcd_data
    },
};

static void ____disp_lcd_dev_data(struct disp_vsync_info *vsync,
                                void *dev_par, struct disp_syncgen_par *sgpar)
{
    struct nxp_lcd_plat_data *plcd = &lcd_data;

    struct disp_lcd_param *dst = (struct disp_lcd_param *)plcd->dev_param;

    struct disp_lcd_param *src = dev_par;

    if (src) {

        SET_PARAM(src, dst, lcd_format);

        SET_PARAM(src, dst, lcd_mpu_type);

        SET_PARAM(src, dst, invert_field);

        SET_PARAM(src, dst, swap_RB);

        SET_PARAM(src, dst, yc_order);

        SET_PARAM(src, dst, lcd_init);

        SET_PARAM(src, dst, lcd_exit);
    }

    if (sgpar)
        plcd->sync_gen = sgpar;

    SET_VSYNC_INFO(vsync, plcd->vsync);
}

#else
#define disp_lcd_dev_data(s, p, g)

```

```
#endif /* LCD */
```

LCD 硬件的配置信息，这些参数要根据具体的 LCD 屏进行设置。

#### 四、实验步骤一：LCD 显示实验

##### 1. 编译：

1) 设置工作环境：

```
$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
```

```
$ mkdir -p /usr/local/src/s6818/project
```

2) 部署实验源码，将光盘：05- 实验例程/ 第 13 章/ 13.5-lcd\_test 文件夹拷贝到 /usr/local/src/s6818/project 路径下；

3) 编译并拷贝 lcd\_test 程序：

```
$ cd /usr/local/src/s6818/project/13.5-lcd_test
```

```
$ make
```

```
$ make install
```

```
$ make clean
```

可执行文件会自动拷贝到/opt/tftp 目录下。

##### 2. 运行：

1) 正确设置 ubuntu 系统的网络，保证网络通信正常

2) 准备好 s6818 实验平台，确保已经按照第 11 章节固化好 Linux 操作系统。

*附注：确保交叉网线和交叉串口线已经连接好主机和实验平台，在终端上用“ifconfig eth0 192.168.0.xxx ”命令设置 ip 地址。核对主机网卡的 ip 地址和 s6818 实验平台的 ip 地址为同一个网段。本实验测试时，主机网卡的 ip 地址为 192.168.0.205，s6818 实验平台的 ip 地址为 192.168.0.101。*

3) 运行 minicom 串口终端，给实验平台加电，进入 Linux 系统，可以看到串口终端的启动打印信息。

4) 系统启动完成后，在 minicom 下执行以下命令将 lcd\_test 程序下载到 s6818 实验平台。

```
# tftp -g 192.168.0.205 -r./lcd_test -l/home/app/lcd_test
```

5) 给 lcd\_test 添加可执行权限

```
#cd /home/app/
```

```
# chmod 777 lcd_test
```

6) 运行 lcd\_test

```
$ ./lcd_test
```

```
Framebuffer:1024 * 600
```

```
Bits: 32
```

```
Draw retangle
```

```
Draw square Draw circle
```

运行 lcd\_test 程序后，可以在显示屏上观察到每隔 5 秒钟依次画出长方形，正方形，圆形。

## 五、实验内容二：触摸屏采集实验

### ● 电容式触摸屏简介

电容式触摸屏是利用人体的电流感应进行工作的。电容式触摸屏的感应屏是一块四层复合玻璃屏，玻璃屏的内表面和夹层各涂有一层导电层，最外层是一薄层砂土玻璃保护层。当我们用手指触摸在感应屏上的时候，人体的电场让手指和和触摸屏表面形成一个耦合电容，对于高频电流来说，电容是直接导体，于是手指从接触点吸走一个很小的电流。这个电流分从触摸屏的四角上的电极中流出，并且流经这四个电极的电流与手指到四角的距离成正比，控制器通过对这四个电流比例的精确计算，得出触摸点的位置。

### ● GSL1680 硬件结构及特性

GSL1680 采用了独特的互电容感应技术，它可以在 1ms 内测量多达 192 个节点。GSL1680 支持广泛的传感器选择，包括单层或双层 ITO，玻璃或薄膜，条形或菱形或其变种传感器图形，薄或厚的 ITO 层。

GSL1680 配备功能强大的 32 位 RISC CPU，可以准确的估计高达 10 个手指触摸区域大小和触摸中心，实现零延迟的手指跟踪，极度柔软舒适的触控感觉。

如下是 GSL1680 的 Touch 接口功能框图如下：

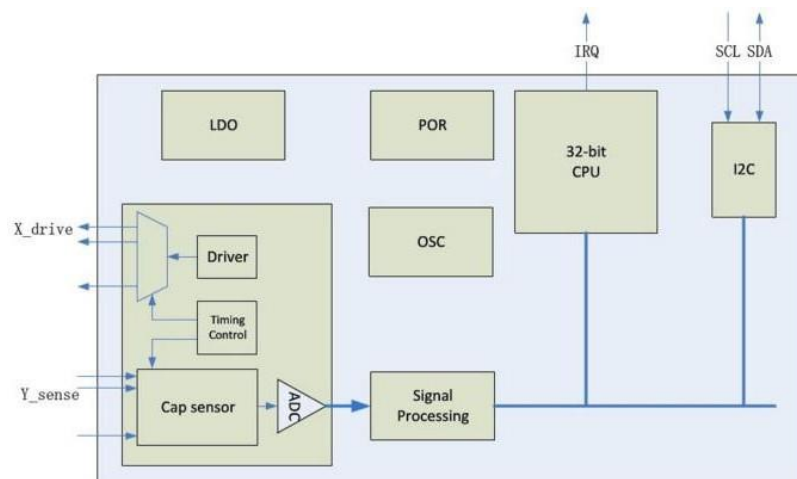


图 3-3 Touch 接口功能框图

- 外部接口

GSL1680 内部 Touch 外部接口如下图所示：

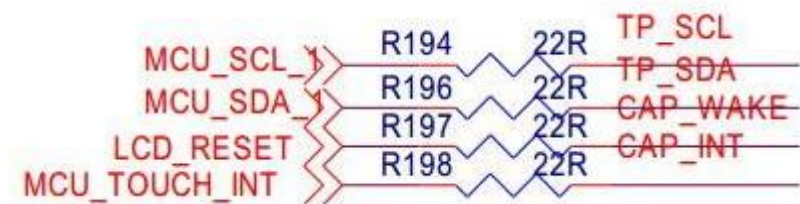


图 3-4 Touch 外部接口

从 GSL1680 的 datasheet 中，我们可以看到，GSL1680 工作在 I2C 的接口方式，从硬件的接口设计上来说，这下面几个控制口，都是需要接的。

1: IRQ 引脚，这个脚是一个中端信号，它用来通知 HOST 端 GSL1680 已经准备好，可以进行读操作了。

2: ~SHUTDOWN 引脚：这个功能主要的作用是将 GSL1680 从睡眠状态转换到工作状态。根据 GSL1680 数据手册上的指令，我们先了解下驱动如何实现电容屏的多点触摸，其实很简单，

主要需要触摸屏 IC GSL1680 能够捕获多点数据，一般电容屏基本都能支持到捕获 2 点以上，而 GSL1680 可以捕获多达 10 个触摸点，编写驱动时，只要去获取这几个点的数据，然后上报就可以了。

- GSL1680 触摸屏驱动关键代码解析

input.h 中定义了 struct input\_dev 结构，它表示 Input 驱动程序的各种信息，对于 Event 设备分为同步设备、键盘、相对设备（鼠标）、绝对设备（触摸屏）等。input\_dev 中定义并归纳了各种设备的信息，例如按键、相对设备、绝对设备、杂项设备、LED、声音设备，强制反馈设备、开关设备等。GSL1680 驱动首先要做的事件就是注册平台设备和驱动程序。

```
static int ____init gsl_ts_init(void)
{
    int ret;

    if(strcasecmp(tp_name, "gslx680-linux") == 0)
    {
        printk("Initial gslx680-linux Touch Driver\n");

        REPORT_DATA_ANDROID_4_0 = 0;
```



```

        is_linux = 1;

        MAX_FINGERS = 1;

        MAX_CONTACTS = 1;
    }

    else if(strcasecmp(tp_name, "gslx680") == 0)
    {

        printk("Initial gslx680 Touch Driver\n");REPORT_DATA_ANDROID_4_0 = 1;

        is_linux = 0;

        MAX_FINGERS = 10;

        MAX_CONTACTS = 10;

    }

    else
    {

        return 0;

    }

    //printk("==gsl_ts_init==\n");

    ret = i2c_add_driver(&gsl_ts_driver);

    printk("==gsl_ts_init== ret=%d\n",ret);return ret;

}

```

平台驱动注册之后，内核就会调用 `gsl_ts_probe()`函数。

```

static int____devinit gsl_ts_probe(struct i2c_client *client,const struct

        i2c_device_id *id)

{

    struct gsl_ts *ts;

    int rc;

```

```

//printk("GSLX680 Enter %s\n", ____func__);

if (!i2c_check_functionality(client->adapter, I2C_FUNC_I2C)) { dev_err(&client-
    >dev, "I2C functionality not supported\n");return -ENODEV;
}

ts = kzalloc(sizeof(*ts), GFP_KERNEL);

if (!ts)
    return -ENOMEM;

printk("==kzall

oc

success=\n");

ts->client = client;

i2c_set_clientdata(client, ts);

ts->device_id = id->driver_data;

gslX680_init();

if(init_chip(ts->client) < 0)
    return -1;

rc = gslX680_ts_init(client, ts);

if (rc < 0) {
    dev_err(&client->dev, "GSLX680 init failed\n");
    goto error_mutex_destroy;
}

gsl_client = client;

```

```

rc= request_irq(client->irq, gsl_ts_irq, IRQF_TRIGGER_RISING,

client->name, ts);if (rc < 0) {

    printk( "gsl_probe: request irq failed\n");

    goto error_req_irq_fail;

}

/* create debug attribute */

//rc = device_create_file(&ts->input->dev, &dev_attr_debug_enable);

#ifdef GSL_MONITOR

printk( "gsl_ts_probe () : queue gsl_monitor_workqueue\n");

INIT_DELAYED_WORK(&gsl_monitor_work, gsl_monitor_worker);

gsl_monitor_workqueue =

create_singlethread_workqueue("gsl_monitor_workqueue");

```

i2c\_add\_driver(&gsl\_ts\_driver);注册触摸屏驱动后 `gsl_ts_probe()` 函数就会调用完成初始化工作。`gsl_ts_probe` 完成初始化后当有触摸屏点击事件时，就会触发触摸屏中断处理函数

```

static irqreturn_t gsl_ts_irq(int irq, void *dev_id)
{

    struct gsl_ts *ts = dev_id;

    print_info("=====gslX680 Interrupt=====\\n");

    disable_irq_nosync(ts->irq);

    if (!work_pending(&ts->work))

    {

        queue_work(ts->wq, &ts->work);    //调用触摸屏工作线程，处理触摸事件

    }

}

```

```
    return IRQ_HANDLED;

}
```

gslX680\_ts\_worker()触摸屏工作线程实现如下。

```
static void gslX680_ts_worker(struct work_struct *work)
{
    struct gsl_ts *ts = container_of(work,
    struct gsl_ts, work);

    int rc, i;

    u8 id, touches;

    u16 x, y;

#ifdef GSL_NOID_VERSION

    u32

    tmp1;

    u8 buf[4] = {0};

    struct gsl_touch_info cinfo;

    memset(&cinfo, 0, sizeof(struct gsl_touch_info));#endif

    print_info("====gslX680_ts_worker====\n");

#ifdef GSL_MONITOR if(i2c_lock_flag != 0)

        goto i2c_lock_schedule;

    else

        i2c_lock_flag = 1;

#endif

    rc = gsl_ts_read(ts->client, 0x80, &ts->touch_data[0], 4);
```

```
if (rc < 0)
{
    dev_err(&ts->client->dev, "read failed\n");goto schedule;
}

touches = ts->touch_data[ts->dd->touch_index];

if(touches > 0)

    gsl_ts_read(ts->client, 0x84, &ts->touch_data[4], 4);

if(touches > 1)

    gsl_ts_read(ts->client, 0x88, &ts->touch_data[8], 4);

if(touches > 2)

    gsl_ts_read(ts->client, 0x8c, &ts->touch_data[12], 4);

if(touches > 3)

    gsl_ts_read(ts->client, 0x90, &ts->touch_data[16], 4);

if(touches > 4)

    gsl_ts_read(ts->client, 0x94, &ts->touch_data[20], 4);

if(touches > 5)

    gsl_ts_read(ts->client, 0x98, &ts->touch_data[24], 4);

if(touches > 6)

    gsl_ts_read(ts->client, 0x9c, &ts->touch_data[28], 4);

if(touches > 7)

    gsl_ts_read(ts->client, 0xa0, &ts->touch_data[32], 4);

if(touches > 8)

    gsl_ts_read(ts->client, 0xa4, &ts->touch_data[36], 4);

if(touches > 9)

    gsl_ts_read(ts->client, 0xa8, &ts->touch_data[40], 4);

print_info("-----touches: %d ---- \n", touches);
```

```

#ifdef

    GSL_NOI

    D_VERSI

    ON

    cinfo.finge

    r_num =

    touches;
    print_info("tp-gsl  finger_num = %d\n",cinfo.finger_num);
    for(i = 0; i < (touches < MAX_CONTACTS ? touches : MAX_CONTACTS); i++)
    {

        cinfo.x[i] = join_bytes( ( ts->touch_data[ts->dd->x_index + 4 * i + 1] & 0xf),ts-
            >touch_data[ts->dd->x_index + 4 * i]);

        cinfo.y[i] = join_bytes(ts->touch_data[ts->dd-
            >y_index + 4 * i + 1],ts-
            >touch_data[ts->dd->y_index + 4 * i
            ]);

        cinfo.id[i] = ((ts->touch_data[ts->dd->x_index + 4 * i + 1] & 0xf0)>>4);
        print_info("tp-gsl  before: x[%d] = %d, y[%d] = %d, id[%d] = %d
            \n",i,cinfo.x[i],i,cinfo.y[i],i,cinfo.id[i]);
    }

    cinfo.finger_num=(ts->touch_data[3]<<24)|(ts->touch_data[2]<<16)

        |(ts->touch_data[1]<<8)|(ts->touch_data[0]);

    gsl_alg_id_main(&cinfo);

    tmp1=gsl_mask_tiaoping();

    print_info("[tp-gsl] tmp1=%x\n",tmp1);

    if(tmp1>0&&tmp1<0xfffffffff)
    {

        buf[0]=0xa;buf[1]=0;

```

```

        buf[2]=0;buf[3]=0;

        gsl_ts_write(ts-
>client,0xf0,buf,4);

        buf[0]=(u8)(tmp1 &
0xff);
        buf[1]=(u8)((tmp1>>8) & 0xff);
        buf[2]=(u8)((tmp1>>16) & 0xff);

        buf[3]=(u8)((tmp1>>24) & 0xff);

        print_info("tmp1=%08x,buf[0]=%02x,buf[1]=%02x,buf[2]=%02x,buf[3]=%02x\n",

            tmp1,buf[0],b
uf[1],buf[2],buf[3]);

        gsl_ts_write(ts-
>client,0x8,buf,4);

    }

```

```

        touches = cinfo.finger_num;#endif

        for(i = 1; i <= MAX_CONTACTS; i++)
        {
            if(touches == 0)
            id_sign[i] = 0;
id_state_flag[
            i] = 0;
        }

        for(i= 0;i < (touches > MAX_FINGERS ? MAX_FINGERS : touches);i++)
        {

#ifdef GSL_NOID_VERSION

            id = cinfo.id[i];
            x =    cinfo.x[i];

            y

```

```

=

cinfo

.y[i];

#else

    x = join_bytes( ( ts->touch_data[ts->dd->x_index + 4 * i + 1] & 0xf),ts-
                    >touch_data[ts->dd->x_index + 4 * i]);

    y = join_bytes(ts->touch_data[ts->dd->y_index + 4 * i + 1],ts-
                    >touch_data[ts->dd->y_index + 4 * i]);

    id = ts->touch_data[ts->dd->id_index + 4 * i] >> 4;#endif

    if(1 <=id && id <= MAX_CONTACTS)
    {

#ifdef FILTER_POINT filter_point(x, y ,id);
#else

        record_point(x, y , id);

#endif

        report_data(ts, x_new, y_new, 10, id);

        id_state_flag[id] = 1;

    }
}

for(i = 1; i <= MAX_CONTACTS; i++)
{

    if( (0 == touches) || ((0 != id_state_old_flag[i]) && (0 == id_state_flag[i])) )

    {

        if(REPORT_DATA_ANDROID_4_0 > 0)
{

            input_report_abs(ts->input,

                            ABS_MT_TOUCH_MAJOR, 0);

```



```

        input_mt_sync(ts->input);

    }

    if(is_linux > 0)

    {

        report_data(ts, x_new, y_new, 0, i);

    }

    id_sign[i]=0;

}

id_state_old_flag[i] = id_state_flag[i];
}

if(0 == touches)

{

    if(REPORT_DATA_ANDROID_4_0 > 0)

        input_mt_sync(ts->input);

#ifdef HAVE_TOUCH_KEY
    if(key_state_flag)

    {

        input_report_key(ts->input, key, 0);input_sync(ts->input);

        key_state_flag = 0;

    }

#endif

}

input_sync(ts->input);

schedule:

#ifdef GSL_MONITOR

    i2c_lock_flag = 0;

    i2c_lock_schedule:

#endif

    enable_irq(ts->irq);                                     // 启动触摸中断

```

|   |
|---|
| } |
|   |

## 六、实验步骤二：触摸屏采集实验

### ● 编译：

1) 设置工作环境：

```
$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
```

```
$ mkdir -p /usr/local/src/s6818/project
```

2) 部署实验源码，将光盘：05-实验例程/第 13 章/13.6-touch\_test 文件夹拷贝到/usr/local/src/ s6818/project 路径下；

3) 编译并拷贝 touch\_test 程序：

```
$ cd /usr/local/src/s6818/project/13.6-touch_test
```

```
$ make
```

```
$ make install
```

```
$ make clean
```

可执行文件会自动拷贝到/opt/tftp 目录下。

### ● 运行：

1) 正确设置 ubuntu 系统的网络，保证网络通信正常

2) 准备好 s6818 实验平台，确保已经按照第 11 章节固化好 Linux 操作系统。

*附注：确保交叉网线和交叉串口线已经连接好主机和实验平台，在终端上用“ifconfig eth0 192.168.0.xxx”命令设置 ip 地址。核对主机网卡的 ip 地址和 s6818 实验平台的 ip 地址为同一个网段。本实验测试时，主机网卡的 ip 地址为 192.168.0.205，s6818 实验平台的 ip 地址为 192.168.0.101。*

3) 运行 minicom 串口终端，给实验平台加电，进入 Linux 系统，可以看到串口终端的启动打印信息。

4) 系统启动完成后，在 minicom 下执行以下命令将 touch\_test 程序下载到 s6818 实验平台。

```
# tftp -g 192.168.0.205 -r./touch_test -l/home/app/touch_test
```

5) 给 touch\_test 添加可执行权限：

```
#cd /home/app/
```

```
# chmod 777 touch_test
```

6) 运行 touch\_test 并点击触摸屏可以在 minicom 下看到触摸点的坐标信息。

```
$ ./touch_test
```

```
event = X, value = 761
event = Y, value = 234
event = X, value = 761
event = Y, value = 234
event = X, value = 467
event = Y, value = 285
event = X, value = 466
event = Y, value = 285
event = X, value = 466
event = Y, value = 285
```

## 七、实验任务

本实验任务需要在实验箱 LCD 和触摸屏上实现铅笔画，也就是手指在屏幕上滑动时，屏幕上能画出相应的曲线来。

提示：需编写通过像素点实现的画线函数。

## 实验四 QT 移植与开发实验

### 一、实验目的

- 了解什么是 QT、为什么选择 QT 以及 QT 的优势具体体现在哪些方面；
- 掌握在主机上编译 Qt5.4 的实验步骤以及如何验证主机成功编译 Qt5.4。
- 掌握 Qt5.4 移植到硬件平台过程中如何编译 QT5.4 源码包；
- 掌握 Qt5.4 移植到硬件平台过程中 QT5.4 文件系统的制作。
- 重点掌握 Linux 下 Qt Creator 的配置以及 Qt Creator 开发 QT 下的应用程序。
- 分别运行桌面版和实验平台上的小程序，验证 QT 的跨平台特点。

### 二、实验环境

- 硬件：电脑（推荐：主频 2GHz+，内存：1GB+）s6818 系列实验平台；
- 软件：ZEmberOS 操作系统。

### 三、实验内容一：主机编译 QT

#### 1. 什么是 QT

Qt 是挪威的 Trolltech 公司基于 C++ 的 GUI 开发工具。QT/X11 和 QTE（QT Embedded）是它其中的两个版本。Qt/X11 是基于 X Windows 系统的 Qt 版本，KDE 便是基于它来构建的。为了适用于嵌入式系统，该公司将 Qt/X11 进行了裁减，发布了 QTE（QT Embedded）版本。QTE 直接基于 Linux 中的 FrameBuffer 设备，删除了 Qt/X11 中一些对资源要求很高的类实现。所以，基于 QTE 实现的应用，不作修改重新编译后，就可以在 Qt/X11 上运行，而反过来便不可以。QPE 是 Trolltech 公司所推出的针对 PDA 软件的整体解决方案，包含了从底层的 GUI 系统、Window Manager、Soft Keyboard 到上层的 PIM、浏览器、多媒体等方面。目前 QPE 的高版本已更名为 Qtopia，其包含了更多功能。

#### 2. 为什么选择 QT

QT 是基于 C++ 的一种语言

相信 C/C++ 目前还是一种很多人都在学习的语言。QT 的好处就在于 QT

本身可以被称作是一种 C++ 的延伸。QT 中有数百个 class 都是用 C++ 写出来的。这也就是说，QT 本身就具备了 C++ 的快速、简易、Object-Oriented Programming（OOP）等等无数的优点。

QT 具有非常好的可移植性（Portable）

QT 不只是可以在 Linux 中运作,也同样可以运行在 Microsoft Windows 中。这也就意味者，利用 QT 编写出来的程式，在几乎不用修改的情况下,就可以同时在 Linux 中和 Microsoft Windows 中运行。QT 的应用非常之广泛，从 Linux 到 Windows 从 x86 到 Embedded 都有 QT 的影子。

#### 四、实验步骤一：主机编译 QT

1)设置工作环境：

```
$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
```

```
$ mkdir -p /usr/local/src/s6818/project
```

2)部署实验源码，将光盘：05-实验例程 / 第 14 章 /14.1-qt\_x86 文件夹拷贝到/usr/local/src/s6818/project 路径下；把 05-实验例程 / 第 14 章 /common 下的 qtbase-opensource-src-5.4.1.tar.gz 拷贝到 /usr/local/src/s6818/project/14.1-qt\_x86 目录；

3)执行下面步骤来进行编译，该编译过程总共大概需要花费 1~6 小时左右（根据硬件配置不同时间不同），请耐心等待：

```
$ cd /usr/local/src/s6818/project/14.1-qt_x86
```

```
$ tar xvf qtbase-opensource-src-5.4.1.tar.gz
```

```
$ cd qtbase-opensource-src-5.4.1
```

```
$ ./configure -opensource -confirm-license -release -prefix ../qt5.4.1 -qt-xcb -no-opengl
```

```
$ make
```

```
$ make install
```

4)编译成功后，在实验目录下 14.1-qt\_x86/qt5.4.1 的 examples 文件夹里有一些实验程序，可以验证 qt 安装是否成功，比如在当前终端执行下列代码进行验证，验证图片如下：

```
$ cd /usr/local/src/s6818/project/14.1-qt_x86
```

```
$ ./qt5.4.1/examples/gui/analogclock/analogclock
```



图 4-1 主机编译 QT 成功界面

## 五、实验内容二：移植 QT 到硬件平台

Qt/Embedded 是一个为嵌入式设备上的图形用户接口和应用开发而订做的 C++ 工具开发包。它通常可以运行在多种不同的处理器上部署的嵌入式 Linux 操作系统上。如果不考虑 X 窗口系统的需要, 基于 Qt/Embedded 的应用程序可以直接对缓冲帧进行写操作。除了类库以外, Qt/Embedded 还包括了几个提高开发速度的工具, 使用标准的 Qt API, 我们可以非常熟练的在 Windows 和 Unix 编程环境里开发应用程序。

Qt/Embedded 提供了一种类型安全的被称之为信号与插槽的真正的组件化编程机制, 这种机制和以前的回调函数有所不同。Qt/Embedded 还提供了一个通用的 widgets 类, 这个类可以很容易的被子类化为客户自己的组件或是对话框。针对一些通用的任务, Qt 还预先为客户定制了像消息框和向导这样的对话框。运行 Qt/Embedded 所需的系统资源可以很小, 相对 X 窗口下的嵌入解决方案而言, Qt/Embedded 只要求一个较小的存储空间 (Flash) 和内存。Qt/Embedded 可以运行在不同的处理器上部署的 Linux 系统, 只要这个系统有一个线性地址的缓冲帧并支持 C++ 的编译器。你可以选择不编译 Qt/Embedded 某些你不需要的功能, 从而大大减小了它的内存占有量。

Qt/Embedded 包括了它自身的窗口系统, 并支持多种不同的输入设备。开发者可以使用他们熟悉的开发环境来编写代码。Qt 的图形设计器 (designer) 可以用来可视化地设计用户接口, 设计器中有一个布局系统, 它可以使你设计的窗口和组件自动根据屏幕空间的大小而改变布局。开发者可以选择一个预定义的视觉风格, 或是建立自己独特的视觉风格。使用 UNIX/LINUX 操作系统的用户, 可以在工作站上通过一个虚拟缓冲帧的应用程序仿真嵌入式系统的显示终端。

Qt/Embedded 也提供了许多特定用途的非图形组件, 例如国际化, 网络和数据库交互组件。

Qt/Embedded 是成熟可靠的工具开发包, 它在世界各地被广泛使用。除了在商业上的许多应用以外, Qt/Embedded 还是为小型设备提供的 Qtopia 应用环境的基础。Qt/Embedded 以简洁的系统, 可视化的表单设计和详致的 API 让编写代码变得愉快和舒畅。

QT/Embedded 是面向嵌入式系统的 QT 版本, QTE 是 Qt/Embedded 的缩写形式。qte 的作用是提供嵌入式开发所需的类库以及链接库, 在开发板上运行 QT 程序就需要这些动态链接库。

Qt/Embedded 只支持鼠标和键盘的操作, 但在大部分嵌入式系统中利用触摸屏, 所以用户必须对触摸屏的相关操作编译成共享库或静态库。

Qt/Embedded 采用两种方式进行发布: 在 GPL 协议下发布的 free 版与专门针对商业应用的 commercial 版本。二者除了发布方式外, 在源码上没有任何区别。纵向看来, 当前主流的版本为 Qtopia 的 2.x 系列与最新的 3.0x 系列。

其中 2.0 版本系统较多地应用于采用 Qtopia 作为高档 PDA 主界面的应用中；3.x 版本系列则应用于功能相对单一，但需要高级 GUI 图形支持的场合，如 Volvo 公司的远程公交信息系统。

3.x 版本系列的 Qt/Embedded 相对于 2.x 版本系统增加了许多新的模块，如 SQL 数据库查询模块等。几乎所有 2.x 版本中原有的类库，在 3.x 版本中都得到极大程度的增强。这就极大地缩短了应用软件的开发时间，扩大了 Qt/Embedded 的应用范围。

在代码设计上，Qt/Embedded 巧妙地利用了 C++ 独有的机制，如继承、多态、模板等，具体实现非常灵活。但其底层代码由于追求与多种系统、多种硬件的兼容，代码补丁较多，风格稍显混乱。

## 六、实验步骤二：移植 QT 到硬件平台

### ● 编译 QT5.4 源码包

1) 设置工作环境：

```
$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
```

```
$ mkdir -p /usr/local/src/s6818/project
```

2) 部署实验源码，将光盘：05-实验例程/第 14 章/14.2-qt\_arm 文件夹拷贝到 /usr/local/src/s6818/project 路径下；把 05-实验例程/第 14 章/common 下的 qtbase-opensource-src-5.4.1.tar.gz 拷贝到 /usr/local/src/s6818/project/14.2-qt\_arm 目录；

3) 执行下面步骤来解压缩源码：

```
$ cd /usr/local/src/s6818/project/14.2-qt_arm
```

```
$ tar zxvf qtbase-opensource-src-5.4.1.tar.gz
```

执行下面步骤来进行编译：

```
$ cp -r linux-arm-generic-g++ qtbase-opensource-src-5.4.1/mkspecs/devices/
```

```
$ cd /usr/local/src/s6818/project/14.2-qt_arm/qtbase-opensource-src-5.4.1
```

```
$ ./configure -opensource -confirm-license -prefix ../qt5.4.1 -device linux-arm-generic-g++  
-device-option CROSS_COMPILE=arm-none-linux-gnueabi- -no-opengl
```

4) 执行 make 命令，编译源码包，执行 make install 完成安装。

```
$ make
```

```
$ make install
```

### ● QT5.4 文件系统的制作

1) 参照 11.6 章节获得 root-mini，并将其复制到当前目录，执行代码如下

所示：

```
$ cd /usr/local/src/s6818/project/14.2-qt_arm
$ cp -r ../11.6-busybox/root-mini ./
$ cp ../11.6-busybox/mkfs ./
```

2) 将编译好的 QT 库及 DEMO 应用拷贝到文件系统中：

```
$ mkdir -p root-mini/opt/qt5.4.1
$ cp -r qt5.4.1/lib root-mini/opt/qt5.4.1/
$ cp -r qt5.4.1/examples root-mini/opt/qt5.4.1/
$ cp -r qt5.4.1/plugins root-mini/opt/qt5.4.1/
```

3) 修改 root-mini/etc/profile 增加 QT 配置信息：

```
$ gedit root-mini/etc/profile
```

4) 在打开的文本编辑器中文修改文件内容如下，然后保存退出。

```
echo "running /etc/profile"
HOSTNAME='/bin/hostname' PS1='[\u@\h \w]\$'
export PS1 HOSTNAME
export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:$PATH
export LD_LIBRARY_PATH=/lib:/usr/lib:/usr/local/lib:/opt/qt5.4.1/lib
export QT_QPA_PLATFORM_PLUGIN_PATH=/opt/qt5.4.1/plugins/platforms
export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0
export QT_QPA_FONTDIR=/opt/qt5.4.1/lib/fonts
export
QT_QPA_GENERIC_PLUGINS=evdevkeyboard,evdevmouse,evdevtouch,evdev:/dev/inp
export QT_QPA_EVDEV_TOUCHSCREEN_PARAMETERS=/dev/input/event1
export QT_PLUGIN_PATH=/opt/qt5.4.1/plugins
```

5) 重新制作 ext4 文件系统，即可生成映像文件 rootfs.img。

```
$ sudo ./mkfs
```

6) 参照 11.6 章节烧写文件系统镜像到实验板中。开机启动后，在终端运行 qt examples 下的示例程序测试 qt 是否安装成功：

```
# /opt/qt5.4.1/examples/gui/analogclock/analogclock
```



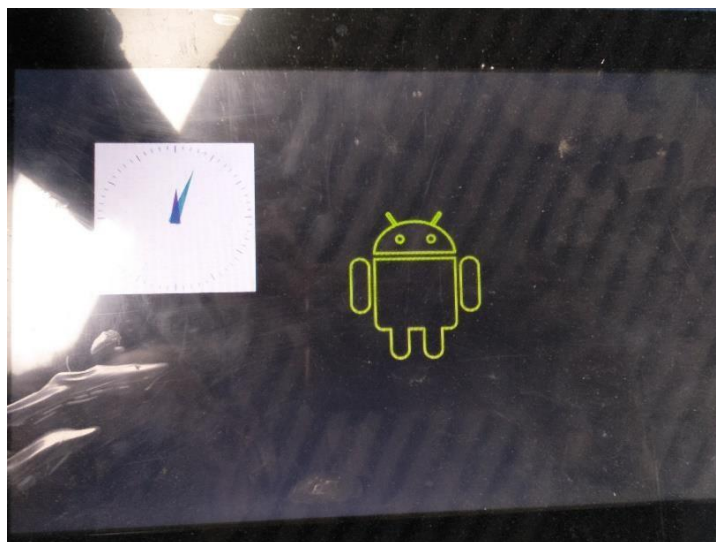


图4-2 实验平台QT成功界面

## 七、实验内容三：QT Creator 开发应用程序

Qt Creator 是跨平台的 Qt IDE，Qt Creator 是 Qt 被 Nokia 收购后推出的一款新的轻量级集成开发环境（IDE）。此 IDE 能够跨平台运行，支持的系统包括 Linux（32 位及 64 位）、Mac OS X 以及 Windows。根据官方描述，Qt Creator 的设计目标是使开发人员能够利用 Qt 这个应用程序框架更加快速及轻易的完成开发任务。

### 功能和优势

QtCreator 主要是为了帮助新 Qt 用户更快速入门并运行项目，还可提高有经验的 Qt 开发人员的工作效率。

#### 1、使用强大的 C++ 代码编辑器可快速编写代码

语法标识和代码完成功能输入时进行静态代码检验以及提示样式上下文相关的帮助代码折叠括号匹配和括号选择模式高级编辑功能

#### 2、使用浏览工具管理源代码

集成了领先的版本控制软件，包括 Git、Perforce 和 Subversion 开放式文件，无须知晓确切的名称或位置搜索类和文件跨不同位置或文件沿用符号在头文件和源文件，或在声明和定义之间切换。

#### 3、为 Qt 跨平台开发人员的需求而量身定制

集成了特定于 Qt 的功能，如信号与槽 (Signals & Slots) 图示调试器，对 Qt 类结构可一目了然集成了 Qt Designer 可视化布局和格式构建器只需单击一下就可生成和运行 Qt 项目

## 八、实验步骤三：QT Creator 开发应用程序

### ● Linux 下 Qt Creator 的安装

### 1) qt Create 下载:

到 qt 官网 [http://download.qt.io/official\\_releases/qtcreator/](http://download.qt.io/official_releases/qtcreator/) 选择一个 qt create 版本下载下来, 以下以 4.7.1 版本为例, ubuntu 下选择后缀为 .run 的文件下载, 此处文件为 qt-creator-opensource-linux-x86\_64-4.7.1.run 下载下来

### 2) 执行下面步骤来进行安装:

```
$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
$ mkdir -p /usr/local/src/s6818/project
$ cd /usr/local/src/s6818/project/14.3-qtCreator
$ chmod a+x qt-creator-opensource-linux-x86_64-4.7.1.run
$ ./qt-creator-opensource-linux-x86_64-4.7.1.run
```

3) 安装过程中, 读者可自定义安装路径, 其余步骤按默认安装即可, 安装完成后, 桌面会出现应用程序的快捷图标

## ● Linux 下 Qt Creator 的配置

1) 在桌面上找到 Qt Creator, 打开后可以看到主界面, 在主界面中从菜单栏点击 "Tools" -> "Options..." -> 在左侧点击 "Kits" -> 选择 "QT Version" 选项卡, 界面如下图所示, 点击按钮添加 Host 版本和 ARM 版本的构建环境路径: (如果 Qt 5.4.1 (qt5.4.1-arm) 前出现一个红色的感叹号, 并且提示没有这个版本的工具链, 执行步骤 2) 后, 红色感叹号自会消失。)

**Qt 5.4.1(qt5.4.1-x86)    /usr/local/src/s6818/project/14.1-qt\_x86/qt5.4.1/bin/qmake**

**Qt 5.4.1(qt5.4.1-arm)    /usr/local/src/s6818/project/14.2-qt\_arm/qt5.4.1/bin/qmake**

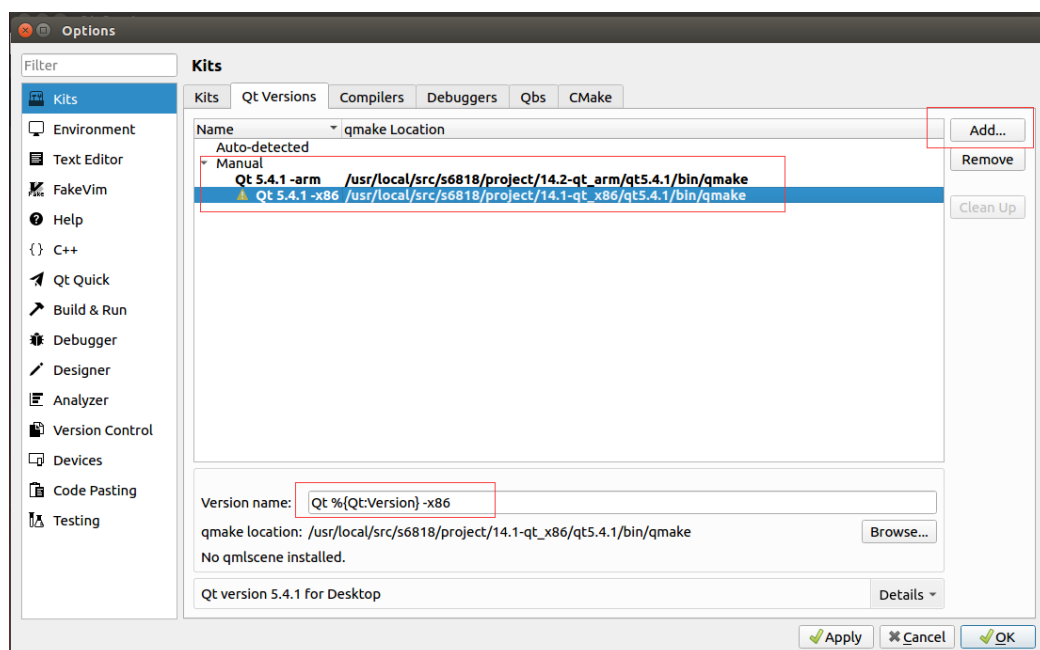


图4-3 手动设置QT版本

2) 接下来将指定编译器, Qt Creator 已经检测到 X86 的 GCC, 只需要手动设置指定用于 arm 开发的交叉工具链即可, 点击“Compilers”选项卡, 点击有操 add 按钮选中 GCC->C 编译器路径为 /usr/local/src/s6818/arm-2009q3/bin/arm-none-linux-gnueabi-gcc, 名称为 arm-gcc 如图中所示。配置完成后点击 Apply, 再点击 ADD->GCC-C++, 编译器路径为/usr/local/src/s6818/arm-2009q3/bin/arm-none-linux-gnueabi-g++, 名称为 arm-g++。配置完成后点击 Apply

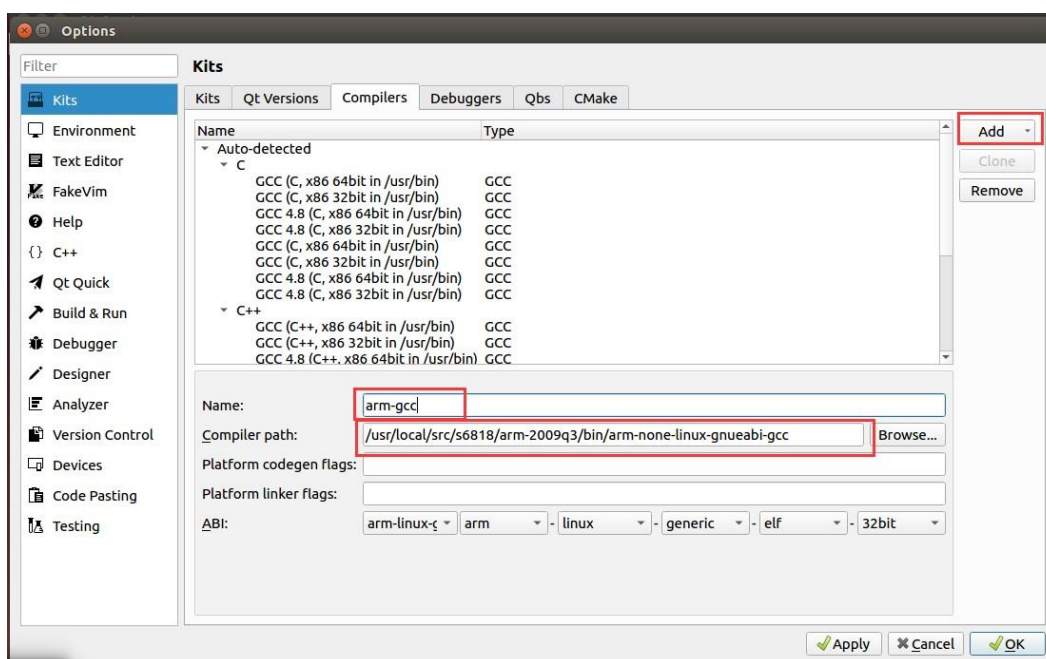
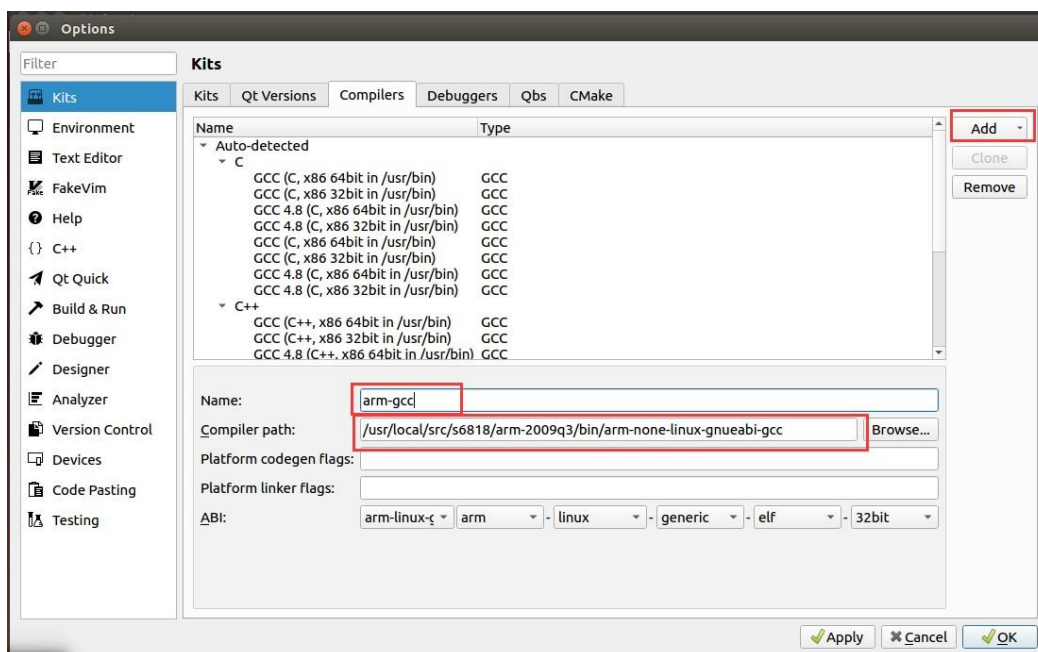


图4-4 手动设置QT工具链

3) 分别对 QT 构建环境与编译器进行绑定：点击“Kits”选项卡，如下图所示，指定设备类型、所用的编译器版本以及 QT 版本，配置桌面和 ARM 两项，分别对应 PC 和 ARM 实验平台两个版本。

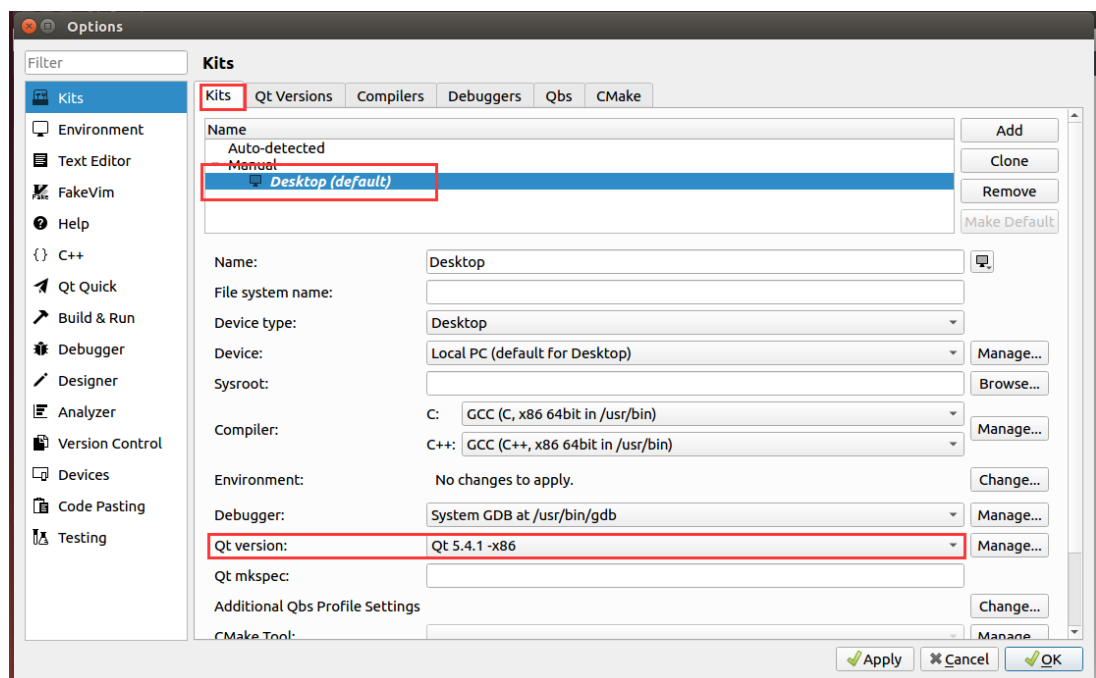


图4-5 QT桌面版本配置

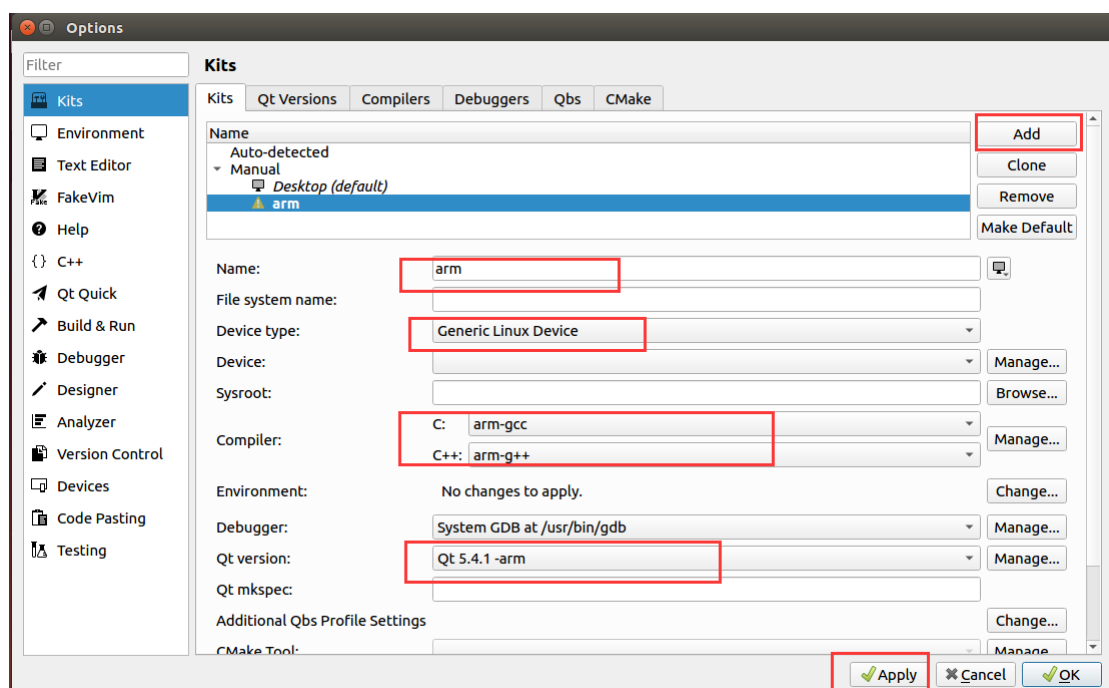


图4-6 QT ARM版本配置

## ● 使用 Qt Creator 进行初步开发

1) 下面新建一个项目，点击菜单“File”->“New File or Project...”，如下图所示，选择“Applications”-“QtWidgets Application”点击 Choose；

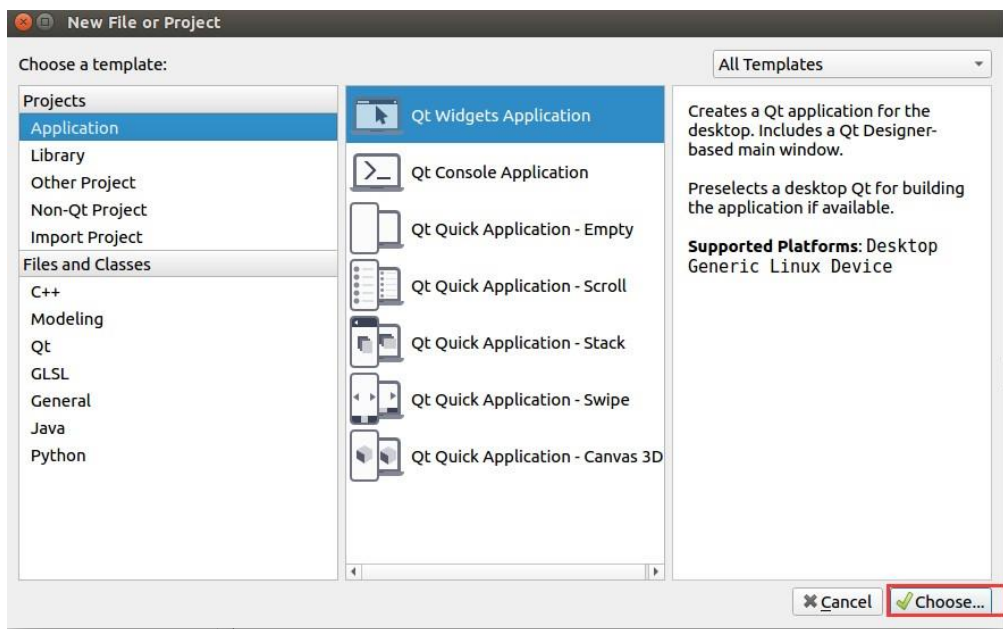


图4-7 新建一个 Qt Gui 应用 HelloWorld

2)输入应用名称， 点击“Next”， 如下图所示：

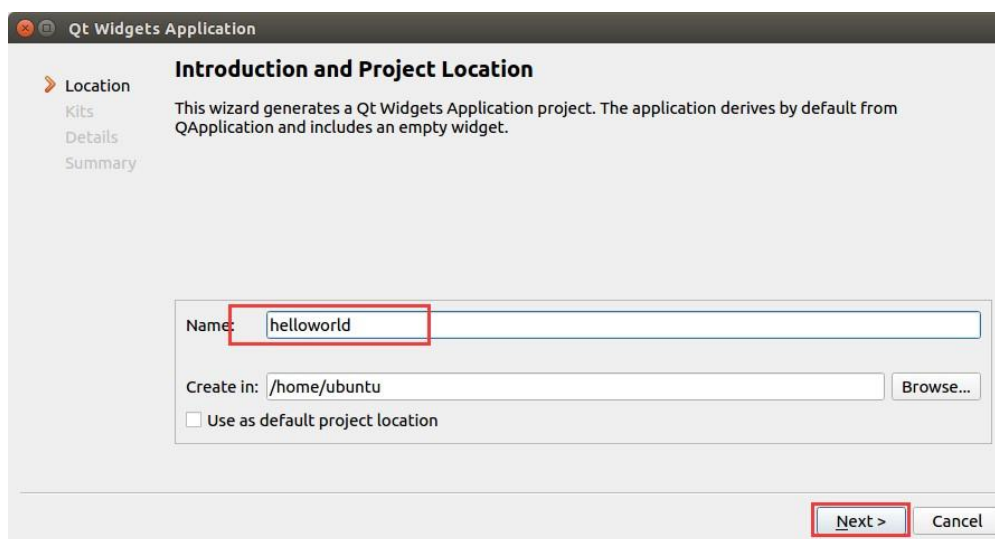


图4-8 选择 helloworld 项目的安装目录

3) 构建平台选择桌面版本：

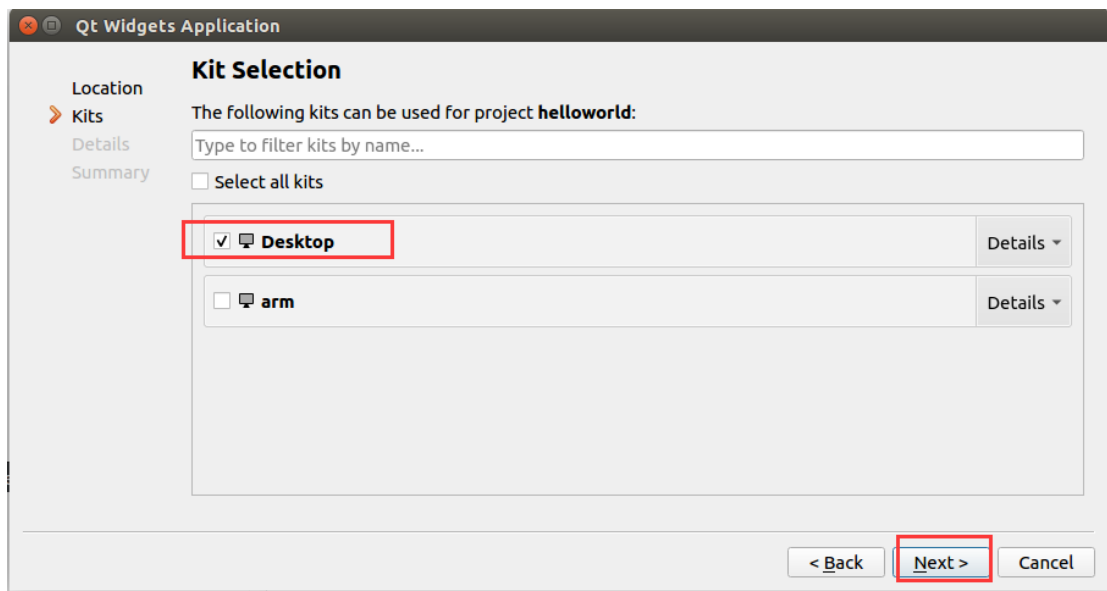


图4-9 选择 桌面套件

4) 类名设置，默认即可，点击 next 按钮

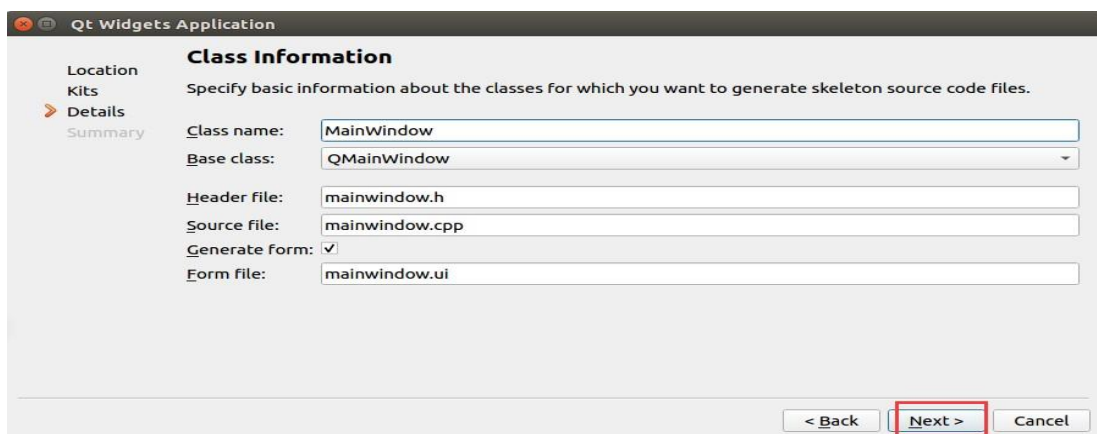
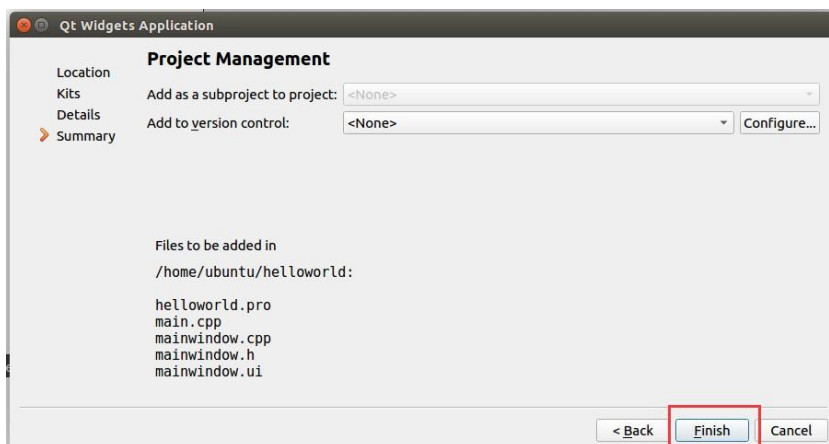


图4-10 类名设置

5) 点击完成按钮完成工程创建



6) 工程新建完成后可以看到工程目录，如下图所示：

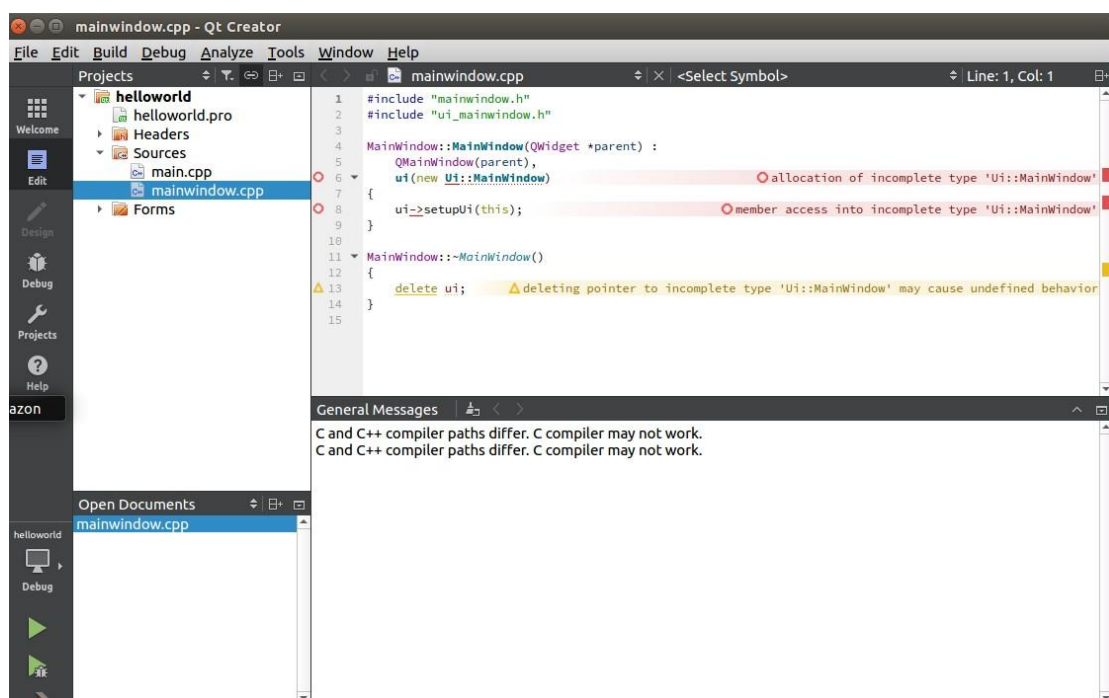
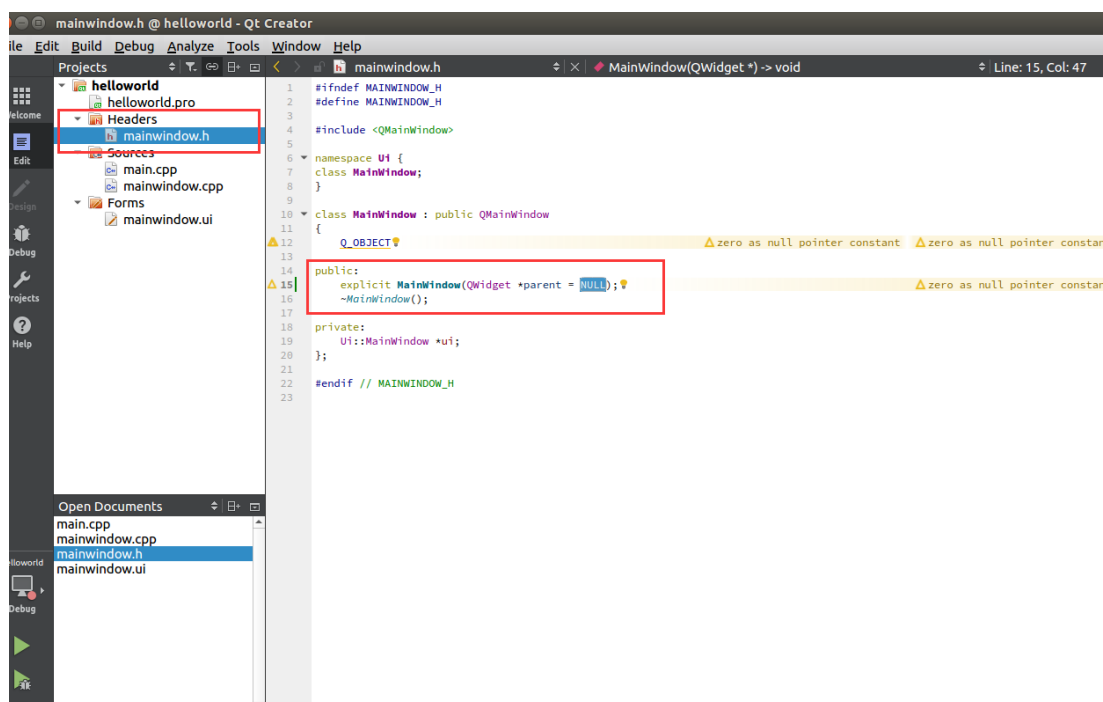


图4-11 helloworld 工程目录文件

7) 修改 mainwindow.h 文件中的 nullptr 为 NULL, 双击 Headers 文件夹下的 mainwindow.h, 修改 15 行内容如下：

explicit MainWindow(QWidget \*parent = NULL);





8) 双击工程目录中“forms”下的 mainwindow.ui 文件，可以看到在 VS 下利用 C#等开发非常类似的界面，可以从左侧选择一些控件添加到工程中，右下角可以针对具体的控件属性进行修改，如下图所示：

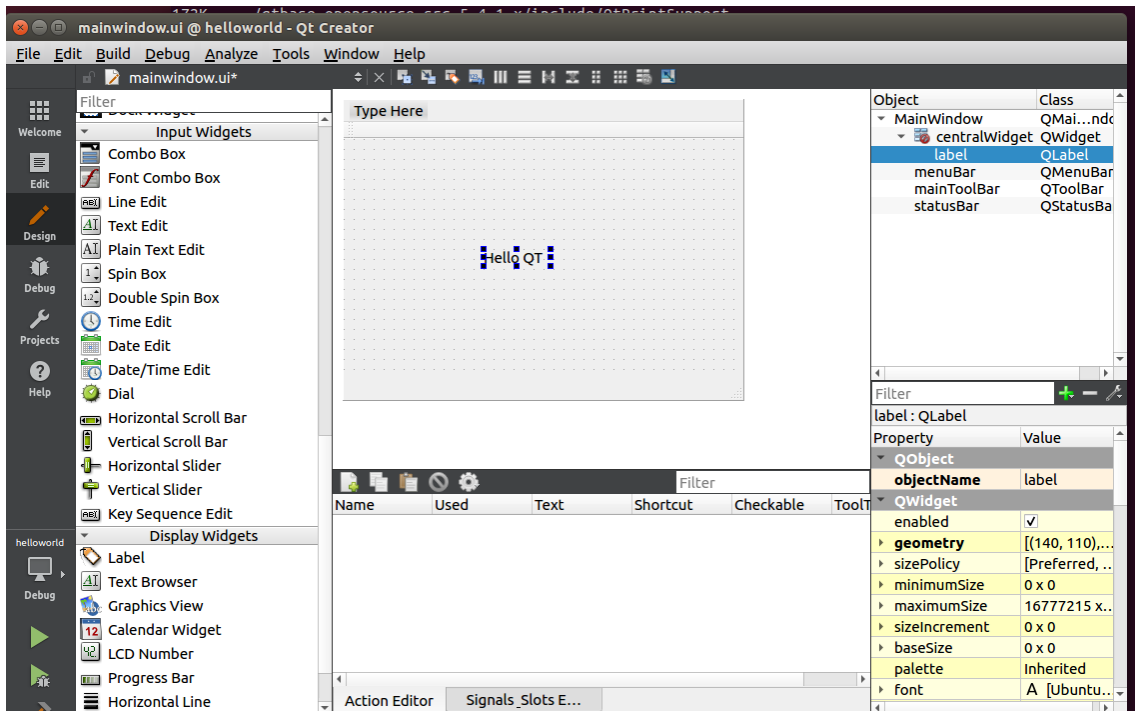



图 4-12 helloworld 工程可视化编辑界面中添加 Lable 控件

9) 点击左下角的  直接运行，构建完成后就会弹出刚刚设计的界面，如下图所示：

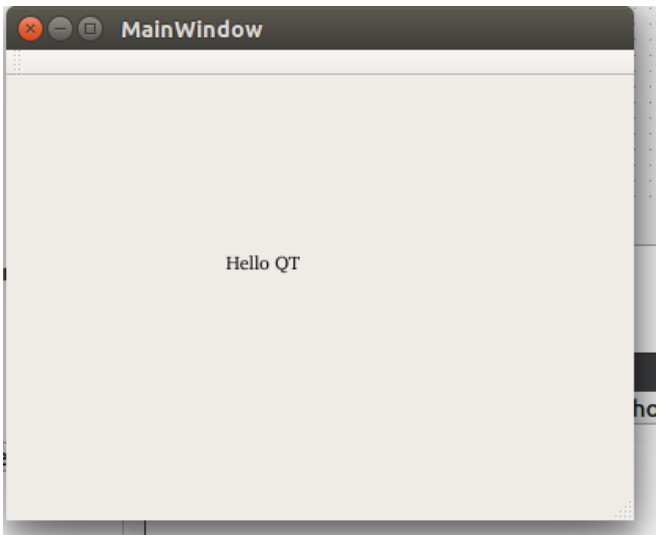


图 4-13 helloworld 工程在 pc 机上运行后的界面

10) 接下来进行实验平台上运行的程序的开发，点击左侧“Projets”按钮，添加 Embedded Linux 的构建设置，环境配置如图所示：



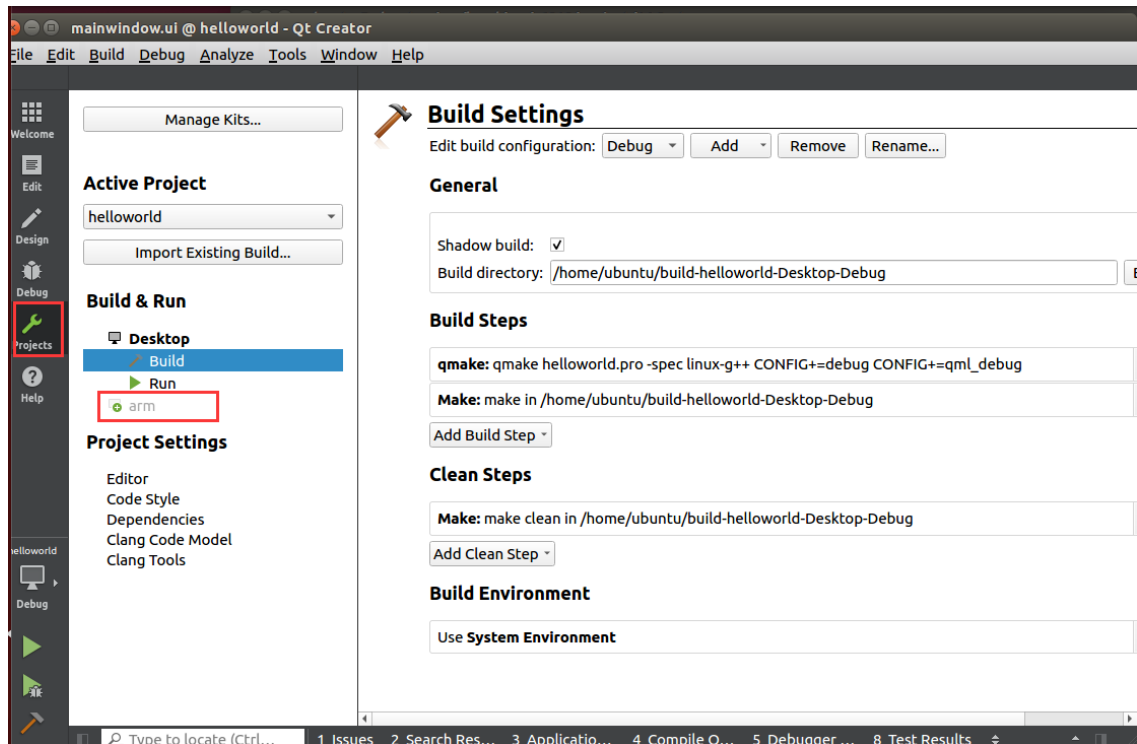
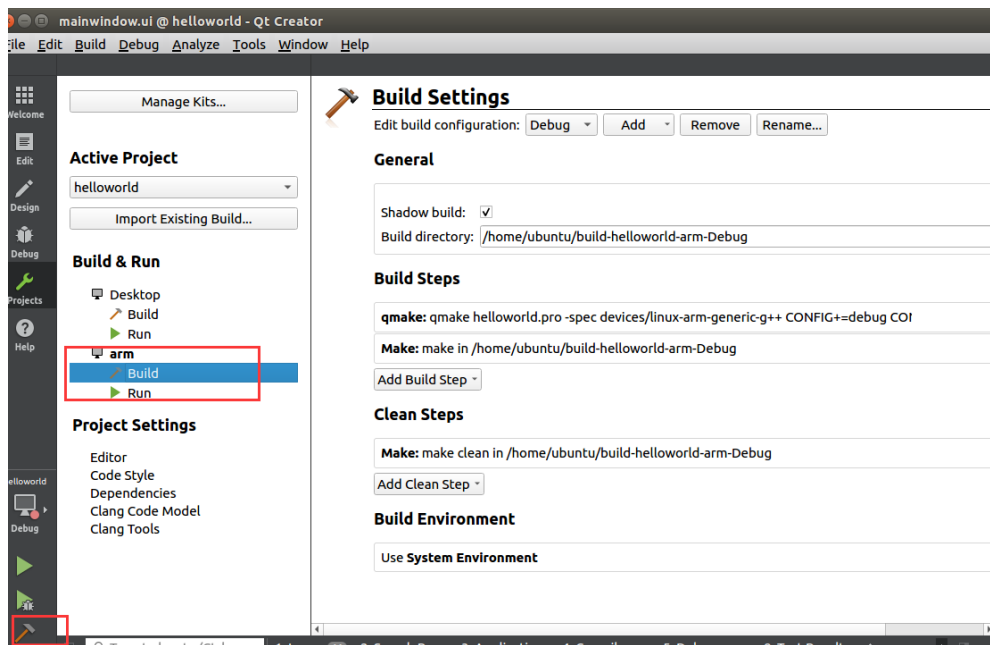


图 4-14 配置 helloworld 工程在实验平台上运行

11) 更改了构建选项后，点击左下角  构建项目按钮，生成 arm 格式的可执行文件。



12) 然后在项目安装目录/home/ubuntu 同等级目录下找到自动生成的文件夹，执行下列命令：

```
$ cd /home/ubuntu
$ cd build-helloworld-arm-Debug
```

```
$ file helloworld
```

如果看到如下信息，则表示 **helloworld** 为在实验平台上可执行的程序。

```
helloworld: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.16, not stripped
```

13) 运行 minicom 串口终端，给实验平台加电，进入 Linux 系统，参照前面章节关于 tftp 的使用，将 **helloworld** 传到实验平台的 **/home/app/** 目录下；

14) 在 minicom 串口终端执行如下命令运行应用程序，在实验平台上可以看到如下界面：

```
# tftp -g 192.168.0.205 -r./helloworld -l/home/app/helloworld
# cd /home/app
# chmod 777 helloworld
# ./helloworld
```

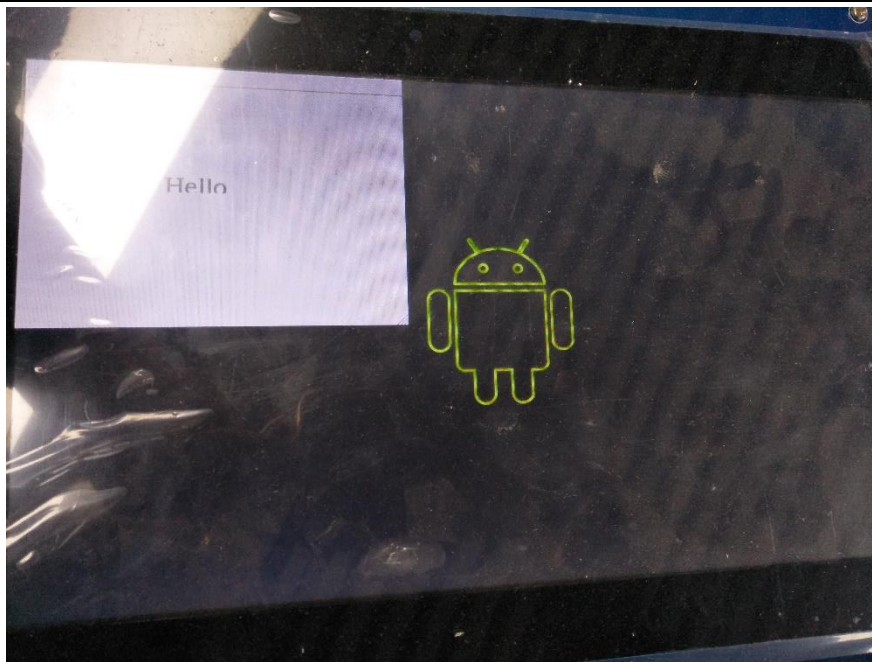


图 4-15 helloworld 工程在实验平台上运行界面

## 九、实验任务

本实验任务是在实验箱上用 QT 控件显示出自己的学号和姓名。

## 实验五 SQLite 数据库实验

### 一、实验目的

- 进一步掌握 linux 下一般应用程序的移植方法实现；
- 通过实验掌握 linux 下 SQLite 数据库的使用。

### 二、实验环境

- 硬件：电脑（推荐：主频 2GHz+，内存：1GB+）s6818 系列实验平台；
- 软件：ZEmberOS 操作系统。

### 三、实验内容

SQLite 是一款轻型的数据库，是遵守 ACID 的关联式数据库管理系统。它的设计目标是嵌入式的，目前已经在很多嵌入式产品中使用。SQLite 占用资源非常的低，在嵌入式设备中，可能只需要几百 K 的内存就够了并且处理速度快。它能够支持 Windows/Linux/Unix 等主流的操作系统。

### 四、实验步骤

- 编译：
  - 1) 设置工作环境：

```
$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH  
$ mkdir -p /usr/local/src/s6818/project
```

2) 部署实验源码，将光盘：05- 实验例程/ 第 12 章/12.10-sqlite 文件夹拷贝到 /usr/local/src/s6818/project 路径下；

3) 进入到 SQLite 实验目录，并解压 SQLite 压缩包：

```
$ cd /usr/local/src/s6818/project/12.10-sqlite  
$ tar zxvf sqlite-3071600.tar.gz
```

4) 建立必要的临时文件夹：

```
$ mkdir sqlite-arm
```

5) 进入 SQLite 目录并进行如下配置：

```
$ cd sqlite-3071600/
```

```
$ ./configure CC=arm-none-linux-gnueabi-gcc --prefix=/usr/local/src/s6818/project/12.10-sqlite/sqlite-arm --disable-tcl --host=arm-linux
```

6) 编译 SQLite:

```
$ make  
$ make install
```

7) 拷贝编译出来的文件到 nfs 目录:

```
$ cd /usr/local/src/s6818/project/12.10-sqlite  
$ cp -r sqlite-arm /opt/nfs
```

## ● 运行:

1) 正确设置 ubuntu 系统的网络, 保证网络通信正常。

2) 准备好 s6818 实验平台, 确保已经按照第 11 章节固化好Linux 操作系统。

*附注: 确保交叉网线和交叉串口线已经连接好主机和实验平台, 在终端上用 “ifconfig ethx 192.168.0.xxx” 命令设置 ip 地址。核对主机网卡的 ip 地址和 s6818 实验平台的 ip 地址为同一个网段。本实验测试时, 主机网卡的 ip 地址为 192.168.0.205, s6818 实验平台的 ip 地址为192.168.0.101。*

3) 运行 minicom 串口终端, 给实验平台加电, 进入 Linux 系统, 可以看到串口终端的启动打印信息。

4) 系统启动完成后, 在minicom 下执行以下命令将 sqlite 下载到 s6818 实验平台:

```
# mount -t nfs 192.168.0.205:/opt/nfs /tmp -o intr,nolock,rsize=1024,wsz=1024  
# cp -r /tmp/sqlite-arm/* /usr/
```

到此, SQLite 数据库移植完成。

下面介绍一些 SQLite 基本命令:

1) 在当前目录下建立或打开 test.db 数据库, 并进入 sqlite 命令终端, sqlite 命令终端以 sqlite>前缀标识:

```
# sqlite3 test.db          # 创建数据库  
  
SQLite version 3.7.16 2013-03-18 11:39:23  
  
Enter ".help" for instructions  
Enter SQL statements  
terminated with a ";"  
  
sqlite>
```

2) 创建名为 film, 并有编号和名称两个列的数据表:

```
sqlite> create table film (number, name);
```

3) 向数据表 film 中添加编号为“1”、名称为“aaa”以及编号为“2”、名称为“bbb”的条目:

```
sqlite> insert into film values (1, 'aaa');  
sqlite> insert into film values (2, 'bbb');
```

4) 查询数据表 film 中存储的内容:

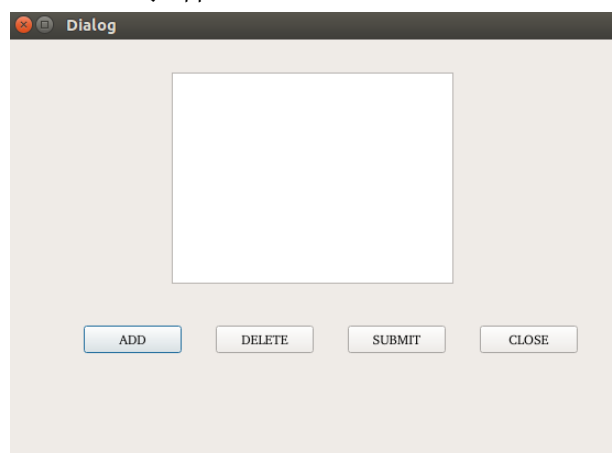
```
sqlite> select * from film;  
1|aaa  
2|bbb
```

5) 退出 SQLite:

```
sqlite> .quit      # 退出sqlite的命令
```

## 五、实验任务

1. 在 test.db 里用 SQLite 再创建一个 student 表, 包括学生学号 ID 和姓名 name 字段, 加入 2 条记录。
2. 按实验四步骤建立好 QT 平台, 设计一个 QT 应用程序, 主界面上显示 “Zhejiang Normal University” 信息, 并且有三个按钮, 分别是 Edit student 表、Edit film 表和 Close。Edit 按钮点击后弹出编辑界面如下图所示, 用 QTableView 控件。



## 实验六 网络服务器 BOA 实验

### 一、实验目的

- 掌握 linux 下 Boa 的特点，功能以及实现；
- 掌握 linux 下的嵌入式 Web 服务器移植。

### 二、实验环境

- 硬件：电脑（推荐：主频 2GHz+，内存：1GB+）s6818 系列实验平台；
- 软件：ZEmberOS 操作系统。

### 三、实验内容

随着 Internet 技术的兴起，在嵌入式设备的管理与交互中，基于 Web 方式的应用成为目前的主流，这种程序结构也就是大家非常熟悉的 B/S 结构，即在嵌入式设备上运行一个支持脚本或 CGI 功能的 Web 服务器，能够生成动态页面，在用户端只需要通过 Web 浏览器就可以对嵌入式设备进行管理和监控，非常方便实用。本节主要介绍这种应用的开发和移植工作。用户首先需要在嵌入式设备上成功移植支持脚本或 CGI 功能的 Web 服务器，然后才能进行应用程序的开发。

#### ● 嵌入式 Web 服务器 Boa 的特点

Boa 是一款单任务的 HTTP 服务器，与其他传统的 Web 服务器不同的是当有连接请求到来时，它并不为每个连接单独创建进程，也不通过复制自身进程来处理多连接，而是通过建立 HTTP 请求列表来处理多路 HTTP 连接请求，同时它只为 CGI 程序创建新的进程，这样就在最大程度上节省了系统资源，这对嵌入式系统来说至关重要。同时它还具有自动生成目录、自动解压文件等功能，因此，Boa 具有很高的 HTTP 请求处理速度和效率，在嵌入式系统中具有很高的应用价值。

#### ● Boa 的功能实现

嵌入式 Web 服务器 Boa 和普通 Web 服务器一样，能够完成接收客户端请求、分析请求、响应请求、向客户端返回请求结果等任务。它的工作过程主要包括：

完成 Web 服务器的初始化工作，如创建环境变量、创建 TCP 套接字、绑定端口、开始侦听、进入循环结构，以及等待接收客户浏览器的连接请求；

当有客户端连接请求时，Web 服务器负责接收客户端请求，并保存相关请求信息；

在接收到客户端的连接请求之后，分析客户端请求，解析出请求的方法、URL

目标、可选的查询信息及表单信息，同时根据请求做出相应的处理；

Web 服务器完成相应处理后，向客户端浏览器发送响应信息，关闭与客户机的 TCP 连接。嵌入式 Web 服务器 Boa 根据请求方法的不同，做出不同的响应。如果请求方法为 HEAD， 则

直接向浏览器返回响应首部；如果请求方法为 GET，则在返回响应首部的同时，将客户端请求的 URL 目标文件从服务器上读出，并且发送给客户端浏览器；如果请求方法为 POST，则将客户发送过来的表单信息传送给相应的 CGI 程序，作为 CGI 的参数来执行 CGI 程序，并将执行结果发送给客户端浏览器。Boa 的功能实现也是通过建立连接、绑定端口、进行侦听、请求处理等来实现的。其初始化部分的源代码如下：

```
int server_s;

server_s = socket( SERVER_PF,SOCK_STREAM,IPPROTO_TCP );

if( server_s == - 1)
{
    DIE( unable to create socket );
}

if( set_nonblock_fd( server_s) == - 1)
{
    DIE( unable to set server socket to nonblocking );
}

if( fcntl( server_s,F_SETFD,1) == - 1)
{
    DIE( can't set close! on! exec on server socket! );
}

if( ( setsockopt( server_s, SOL_SOCKET,SO_REUSEADDR,(
void*)&sock_opt,sizeof( sock_opt) ) ) == - 1)
{
    DIE( setsockopt );
}

if( bind_server( server_s, server_ip, server_port) == - 1)
{
    DIE( unable to bind );
}
```

```

}

if( listen( server_s, backlog) == - 1)
{
    DIE( unable to listen) ;
}

```

上述代码主要用于打开一个有效的 socket 描述符，然后将其转换为无阻塞套接字。函数 bind() 用于建立套接字描述符与指定端口间的关联，并通过函数 listen() 在该指定端口侦听，等待远程连接请求。当侦听到连接请求时，Boa 调用函数 get\_request( int server\_sock) 获取请求信息，通过调用函数 accept() 为该请求建立一个连接。在建立连接之后，接收请求信息，同时对请求进行分析。当有 CGI 请求时，为 CGI 程序创建进程，并将结果通过管道发送输出。

### ● 嵌入式 Web 服务器移植

由于嵌入式设备资源一般都比较有限，并且也不需要能同时处理很多用户的请求，因此不会使用 Linux 下最常用的如 Apache 等服务器，而需要使用一些专门为嵌入式设备设计的 Web 服务器，这些 Web 服务器在存储空间和运行时所占有的内存空间上都会非常适合于嵌入式应用场合。典型的嵌入式 Web 服务器有 Boa (www.boa.org) 和 thttpd (http://www.acme.com/software/thttpd/) 等，它们和 Apache 等高性能的 Web 服务器主要的区别在于它们一般是单进程服务器，只有在完成一个用户请求后才能响应另一个用户的请求，而无法并发响应，但这在嵌入式设备的应用场合里已经足够了。

我们介绍比较常用的 Boa 服务器的移植。

Boa 是一个非常小巧的 Web 服务器，可执行代码只有约 60KB。它是一个单任务 Web 服务器，只能依次完成用户的请求，而不会 fork 出新的进程来处理并发连接请求。但 Boa 支持 CGI，能够为 CGI 程序 fork 出一个进程来执行。Boa 的设计目标是速度和安全，在其站点公布的性能测试中，Boa 的性能要好于 Apache 服务器。

## 四、实验步骤

### ● 编译：

1) 设置工作环境：

```

$ PATH=/usr/local/src/s6818/arm-2009q3/bin:$PATH
$ mkdir -p /usr/local/src/s6818/project

```

2) 部署实验源码，将光盘：05- 实验例程 / 第 12 章 /12.9-boa 文件夹拷贝到/usr/local/src/s6818/project 路径下；



3) 进入到 boa 实验目录，并解压相应工具包及库：

```
$ cd /usr/local/src/s6818/project/12.9-boa  
$ tar zxvf boa-0.94.13.tar.gz
```

4) 配置 boa 生成 Makefile:

```
$ cd boa-0.94.13/src  
$ ./configure --host=arm-linux --target=arm-linux
```

5) 修改源码文件:

修改 12.9-boa /boa-0.94.13/src/Makefile:

修改:

```
CC = gcc
```

```
CPP = gcc -E
```

为:

```
CC = arm-none-linux-gnueabi-gcc
```

```
CPP = arm-none-linux-gnueabi-gcc -E
```

修改 12.9-boa /boa-0.94.13/src/compat.h:

修改:

```
#define TIMEZONE_OFFSET(foo) foo##->tm_gmtoff
```

为:

```
#define TIMEZONE_OFFSET(foo) foo->tm_gmtoff
```

修改 12.9-boa /boa-0.94.13/src/boa.c:

修改:

```
    if (setuid(0) != -1) {  
        DIE("icky Linux kernel bug!");  
    }
```

为:

```
    #if 0  
    if (setuid(0) != -1) {  
        DIE("icky Linux kernel bug!");  
    }  
    #endif
```

修改:

```
if (passwdbuf == NULL) {  
  
    DIE("getpwuid");  
  
}  
  
if (initgroups(passwdbuf->pw_name, passwdbuf->pw_gid) == -1) {  
  
    DIE("initgroups");  
  
}
```

为:

```
#if 0  
  
if (passwdbuf == NULL) {  
  
    DIE("getpwuid");  
  
}  
  
if (initgroups(passwdbuf->pw_name, passwdbuf->pw_gid) == -1) {  
  
    DIE("initgroups");  
  
}  
  
#endif
```

修改/usr/local/src/s6818 /project/12.9-boa/boa-0.94.13/src/log.c:

修改:

```
if (dup2(error_log, STDERR_FILENO) == -1) {  
  
    DIE("unable to dup2 the error log");  
  
}为:  
#if 0  
  
if (dup2(error_log, STDERR_FILENO) == -1) {  
  
    DIE("unable to dup2 the error log");  
  
}  
  
#endif
```

6) 修改完源码文件后输入命令开始编译源码:

```
$ make
```

```
$ arm-none-linux-gnueabi-strip boa
```

7 ) 修 改   boa 配 置 文 件 /usr/local/src/s6818/project/12.9-boa/boa-0.94.13/boa.conf:

修改:

```
User nobody
```

```
Group nogroup
```

为:

```
User 0
```

```
Group 0
```

修改:

```
#ServerName www.your.org.here
```

为:

```
ServerName www.zonesion.com.cn
```

修改:

```
DocumentRoot /var/www
```

为:

```
DocumentRoot /www
```

修改:

```
MimeTypes /etc/mime.types
```

为:

```
MimeTypes /dev/null
```

修改:

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
```

为:

```
ScriptAlias /cgi-bin/ /www/cgi-bin/
```

8) 准备 BOA 实验必备源码并拷贝到/opt/nfs 目录（实验中将通过 ftp 服务加载到实验平台运行的 linux 上）。

Ubuntu Linux 终端:

```
$ cd /usr/local/src/s6818/project/12.9-boa  
$ chmod 777 -R *  
$ cp boa-0.94.13/boa.conf /opt/nfs  
$ cp boa-0.94.13/src/boa /opt/nfs  
$ cp -r www /opt/nfs
```

● 运行:

- 1) 正确设置 ubuntu 系统的网络, 保证网络通信正常。
- 2) 准备好 s6818 实验平台, 确保已经按照第 11 章节固化好Linux 操作系统。

*附注: 确保交叉网线和交叉串口线已经连接好主机和实验平台, 在终端上用 “ifconfig ethx 192.168.0.xxx” 命令设置 ip 地址。核对主机网卡的 ip 地址和 s6818 实验平台的 ip 地址为同一个网段。本实验测试时, 主机网卡的 ip 地址为 192.168.0.205, s6818 实验平台的 ip 地址为192.168.0.101。*

- 3) 运行 minicom 串口终端, 给实验平台加电, 进入 Linux 系统, 可以看到串口终端的启动打印信息。

- 4) 系统启动完成后, 在minicom 下执行以下命令将boa 下载到 s6818 实验平台:

```
# mount -t nfs 192.168.0.205:/opt/nfs /tmp -o intr,nolock,rsize=1024,wsz=1024  
# mkdir -p /etc/boa  
# cp /tmp/boa.conf /etc/boa/  
# mkdir -p /var/log/boa  
# mkdir -p /www/cgi-bin  
# cp -r /tmp/www/* /www
```

- 5) 运行测试BOA 网络服务器:

```
# cd /tmp  
# ./boa
```

在 PC 端 IE 浏览器中输入实验平台的 IP 地址: <http://192.168.0.101/index.html>, 出现欢迎网页, BOA 搭建成功, 其中 192.168.0.101 为实验平台的主板网卡的 IP 地址。

如果页面打开显示的是源文件, 需要使用 IE 浏览器兼容模式下运行。

## 五、实验任务

本实验任务是通过网页实现对实验箱中2个LED灯的亮灭控制。需要利用实验二的led驱动，用insmod命令把led驱动先加载，然后在主机上输入http://192.168.0.101/t-led.html，网页显示如下图所示。

