

Version：V1.0.0

Date：2016.10.08

# 高通lk和gpio配置

Copyright

Copyright ©2016 Fibocom Wireless Inc . All rights reserved.

Without the prior written permission of the copyright holder, any company or individual is prohibited to excerpt, copy any part of or the entire document, or transmit the document in any form.

Attention

The document is subject to update from time to time owing to the product version upgrade or other reasons. Unless otherwise specified, the document only serves as the user guide. All the statements, information and suggestions contained in the document do not constitute any explicit or implicit guarantee.

Trademark

The trademark is registered and owned by Fibocom Wireless Inc.

Versions

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Remarks** |
| V1.0.0 | 2016-10-08 | Initial Version |
|  |  |  |

Applicability Type

|  |  |  |
| --- | --- | --- |
| **No.** | **Type** | **Note** |
|  |  |  |
|  |  |  |

**Content**

[1.1 ：Boot Loader 的概念 4](E:\\部门管理\\FIBOCOM_SC800基于fastboot的下载模式.doc" \l "_Toc459377073)

1.2：bootloader的启动加载模式 [4](#_Toc459377074)

[1.3：常用bootloader 6](#_Toc459377078)

[2 ：little kernel分析 1](#_Toc459377079)3

[Fastboot协议 2](#_Toc459377080)9

[3：gpio配置 3](#_Toc459377079)6

**1.1** Boot Loader 的概念:

简单地说，Boot Loader 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。

Boot Loader为引导加载程序，是系统加电后运行的第一段软件代码，例如PC 机的引导加载程序，包括：BIOS(其本质就是一段固件程序)和位于硬盘 MBR 中的 OS Boot Loader，比如LILO、GRUB 等。BIOS的主要任务是进行硬件检测和资源分配；将MBR中的OS Boot Loader读到系统的 RAM 中；将控制权交给 OS Boot Loader。Boot Loader 的主要运行任务是将内核映象从硬盘上读到 RAM 中和跳转到内核的入口点去运行，也即启动操作系统。

在嵌入式系统中通常并没有像 BIOS 那样的固件程序（注：有的嵌入式 CPU 也会内嵌一段短小的启动程序），整个系统的加载启动任务完全由 Boot Loader 完成。

在嵌入式世界里建立一个通用的 Boot Loader 几乎是不可能的，Boot Loader 对硬件的依赖性非常强，特别是在嵌入式系统世界中尽管如此，仍可对 Boot Loader 归纳出一些通用的概念，以指导用户特定的 Boot Loader 设计与实现。

Boot Loader依赖于CPU 的体系结构和具体的嵌入式板级设备的配置。不同的CPU体系结构都有不同的Boot Loader，有些 Boot Loader 也支持多种CPU体系结构，例如U-Boot同时支持ARM和MIPS体系结构。对于两块不同的嵌入式板，即使它们基于同一种 CPU，要想让运行在一块板子上的 Boot Loader也能运行在另一块板子上，通常也都需要修改 Boot Loader源程序。

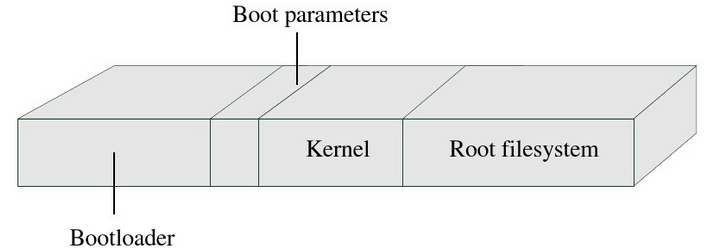
系统加电或复位后，所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。比如，基于 ARM7TDMI core 的 CPU 在复位时通常都从地址 0x00000000 取它的第一条指令。基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备被映射到该预先安排的地址上。比如：ROM、EEPROM 或 FLASH 等。

Boot Loader的启动过程可以是单阶段（Single Stage）或多阶段（Multi-Stage）。一些只需完成很简单功能的boot loader可能是单阶段的。通常多阶段的 Boot Loader 能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的 Boot Loader 大多都是 2 阶段的启动过程，也即启动过程可以分为 stage1 和 stage2 两部分。

Boot Loader的位置：通常固化在硬件上的某个固态存储设备上，加电后自启动。

Boot Loader所在的存储设备的空间分配：





用来控制 Boot Loader 的设备或机制：

主机和目标机之间一般通过串口建立连接，Boot Loader 软件在执行时通常会通过串口来进行 I/O，比如：输出打印信息到串口，从串口读取用户控制字符等。

**1.2**  bootloader的启动加载模式和下载模式

启动加载模式：

这种引导方式也称为“自主（autonomous）引导方式”，也即BootLoader从目标机的某个固态存储设备上将操作系统加载到RAM中并引导运行，整个过程并没有用户的介入。这种引导方式是BootLoader的正常工作模式下普遍采用的一种引导方式。因此在嵌入式产品发布的时候，BootLoader一般以这种引导方式对内核代码进行启动引导。

下载模式：

在这种引导方式下目标机上的BootLoader将通过串口连接或网络连接等通信手段从主机下载文件，如下载应用程序、数据文件、内核映像等。从主机下载的文件通常首先被BootLoader保存到目标机的RAM中，然后再被BootLoader写到目标机上的固态存储设备中，其后完成内核的引导运行。 BootLoader的这种引导方式通常在系统研发和更新时使用。

**1.2.1** BootLoader的基本执行流程

从操作系统的角度看，Boot Loader 的总目标就是正确地调用内核来执行。另外，由于 Boot Loader 的实现依赖于 CPU 的体系结构，因此大多数 Boot Loader 都分为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码，比如设备初始化代码等，通常都放在 stage1 中，而且通常都用汇编语言来实现，以达到短小精悍的目的。

而 stage2 则通常用C语言来实现，这样可以实现给复杂的功能，而且代码会具有更好的可读性和可移植性。

**1.2.2**：

Boot Loader 的 stage1 通常包括以下步骤(以执行的先后顺序)：

1: 硬件设备初始化。

2：为加载 Boot Loader 的 stage2 准备 RAM 空间。

3：拷贝 Boot Loader 的 stage2 到 RAM 空间中。

Boot Loader 的 stage2 通常包括以下步骤(以执行的先后顺序)：

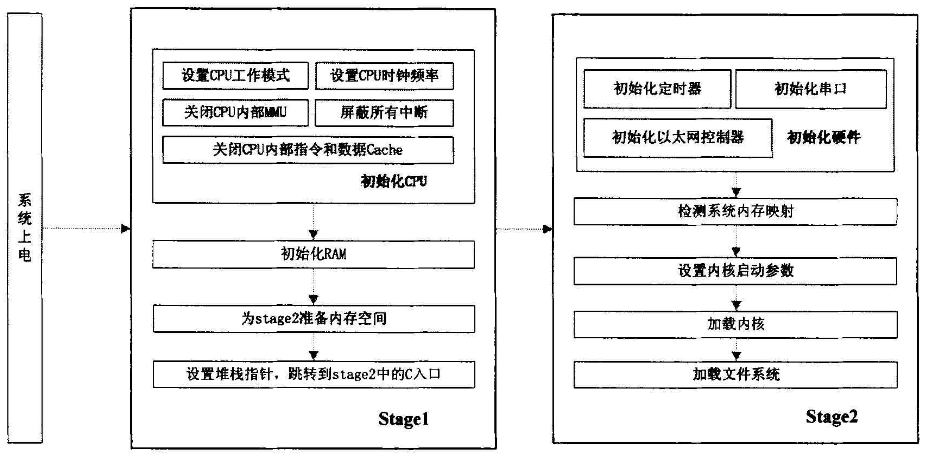
1：初始化本阶段要使用到的硬件设备。

2：检测系统内存映射(memory map)。

3：将 kernel 映像和根文件系统映像从 flash 上读到 RAM 空间中。

为内核设置启动参数。

4：调用内核。



**1.3** **常用的嵌入式 BootLoader**

从上一节的内容可以了解到BootLoader是嵌入式系统中非常重要的一部分，也是系统运行工作的必要组成部分。在嵌入式系统中常见的BootLoader有以下几种。

**1.3.1  U-Boot**

U-Boot是德国DENX小组开发的用于多种嵌入式CPU的BootLoader程序，它可以运行在基于PowerPC、ARM、MIPS等多种嵌入式开发板上。从http://u-boot.sourceforge.net/或ftp://ftp.denx. de/pub/u-boot/站点都可以下载U-Boot的源代码，U-Boot源代码的主要目录解释如下。

  ● board  目标板相关文件，主要包含SDRAM、Flash驱动；

  ● common  独立于处理器体系结构的通用代码，如内存大小探测与故障检测；

  ● cpu  与处理器相关的文件，如mpc8xx子目录下含串口、网口、LCD驱动及中断初始化等文件；

  ● driver  通用设备驱动，如CFI Flash驱动（目前对Intel Flash支持较好）；

  ● doc  U-Boot的说明文档；

  ● examples  可在U-Boot下运行的示例程序，如hello\_world.c、timer.c；

  ● include  U-Boot头文件，尤其是configs子目录下与目标板相关的配置头文件是移植过程中经常要修改的文件；

  ● lib\_xxx  处理器体系相关的文件，如lib\_ppc、lib\_arm目录分别包含与PowerPC、ARM体系结构相关的文件；

  ● net  与网络功能相关的文件目录，如bootp、nfs、tftp；

  ● post  上电自检文件目录，尚有待于进一步完善；

  ● rtc  RTC（Real Time Clock，实时时钟）驱动程序；

  ● tools  用于创建U-Boot S-RECORD和BIN镜像文件的工具。

**1.3.2  VIVI**

VIVI是由韩国MIZI公司开发的专门用于ARM产品线的一种BootLoader。因为VIVI 目前只支持使用串口和主机通信，所以必须使用一条串口电缆来连接目标板和主机。VIVI的源代码下载地址为：http://www.mizi.com/developer/s3c2410x/download/vivi.html，VIVI一般有如下作用。

（1）把内核（kernel）从Flash复制到RAM，然后启动它；

（2）初始化硬件；

（3）下载[程序](http://www.itisedu.com/phrase/200604232224305.html" \t "_new)并写入Flash；

（4）检测目标板。

vivi.tar.bz2源代码包解压后的目录结构如下所示：

# tree –L 1

.

|-- COPYING

|-- CVS

|-- Documentation

|-- Makefile

|-- Rules.make

|-- arch

|-- drivers

|-- include

|-- init

|-- lib

|-- scripts

|-- test

`-- util

10 directories, 3 files

其中VIVI主要目录介绍如下。

  ● CVS  存放CVS工具相关的文件；

  ● Documentation  存放一些使用VIVI的帮助文档；

  ● arch  存放一些平台相关的代码文件；

  ● drivers  存放VIVI相关的驱动代码；

  ● include  存放所有VIVI源码的头文件；

  ● init  存放VIVI初始化代码；

  ● lib  存放VIVI实现的库函数文件；

  ● scripts  存放VIVI脚本配置文件；

  ● test  存放一些测试代码文件；

  ● util  存放一些NAND Flash烧写image相关的工具实现代码。

**1.3.3  Blob**

Blob是Boot Loader Object的缩写，是一款功能强大的BootLoader。其源码在[http://sourceforge. net/projects/blob](http://sourceforge.net/projects/blob)上可以获取。Blob最初是由Jan-Derk Bakker和Erik Mouw两人为一块名为LART（Linux Advanced Radio Terminal）的开发板写的，该板使用的处理器是StrongARM SA-1100，现在Blob已经被成功地移植到许多基于ARM的CPU上。

**1.3.4  RedBoot**

RedBoot是一个专门为嵌入式系统定制的引导启动工具，最初由Redhat开发，它是基于eCos（Embedded Configurable Operating System）的硬件抽象层，同时它继承了eCos的高可靠性、简洁性、可配置性和可移植性等特点。RedBoot集Bootloader、调试、Flash烧写于一体，支持串口、网络下载，执行嵌入式应用程序。既可以用在产品的开发阶段（调试功能），也可以用在最终的产品上（Flash更新、网络启动）。RedBoot支持下载和调试应用程序，开发板可以通过BOOTP/DHCP协议动态配置IP地址，支持跨网段访问。用户可以通过tftp协议下载应用程序和image，或者通过串口用x-modem/y-modem下载。RedBoot支持用GDB（the GNU debugger）通过串口或者网卡调试嵌入式程序，可对gcc编译的程序进行源代码级的调试。相比于简易jtag调试器，它可靠、高速（CPU的cache打开后，通过网卡tftp下载能达到1Mbps，GDB下载的速度能达到2Mbps）、稳定，用户可通过串口或网卡，以命令行的形式管理Flash上的image，下载image到Flash。动态配置RedBoot启动的各种参数、启动脚本，上电后RedBoot可自动从Flash或tftp服务器上下载应用程序执行。在http://sourceware.org/redboot站点可以下载RedBoot源码，同时可以了解更多的关于RedBoot的详细信息，它在嵌入式系统应用中非常广泛。

**1.3.5  ARMboot**

ARMboot是一个以ARM或StrongARM 为内核CPU的嵌入式系统的BootLoader固件程序，该软件的主要目标是使新的平台更容易被移植并且尽可能地发挥其强大性能。它只基于ARM固件，但是它支持多种类型的启动，比如Flash，网络下载通过bootp、dhcp、tftp等。它也是开源项目，可以从http://www.sourceforge.net/projects/armboot网站获得最新的ARMboot源码和详细资料，它在ARM处理器方面应用非常广泛。

**1.3.6  DIY**

DIY（Do It Yourself），即自己制作。以上U-Boot、VIVI、Blob、RedBoot和ARMboot等成熟工具移植起来简单快捷，但同时它们都存在着一定的局限性，首先是因为它们是面向大部分硬件的工具，所以说在功能上要满足大部分硬件的需求，但一般情况下我们只需要与特定的开发板相关的实现代码，其他型号开发板的实现代码对它来说是没有用的，所以通常它们的代码量较大。其次它们在使用上不够灵活，比如在这些工具上添加自己的特有功能相对比较困难，因为必须熟悉该代码的组织关系，以及了解它的配置编译等文件。用DIY的方式自己编写针对目标板的BootLoader不但代码量短小，同时灵活性很大，最重要的是将来容易维护。所以在实际嵌入式产品开发时大都选择DIY的方式编写BootLoader。

2 :Little Kernel分析

Little Kernel是最近几年才推出的一种Bootloader，并被android系统所接受进入android源码树，大量应用于智能手机与平板电脑。

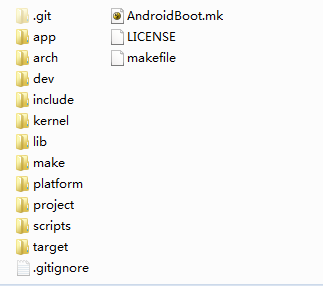
Little Kernel除了支持高通，TI，三星，Telechips等公司的芯片组，目前还具备如下功能：

1. 支持多种NANDFLASH启动。
2. 提供USB驱动，支持开发过程中通过USB升级image文件。
3. 提供键盘驱动，支持开发人员进入fastboot模式升级image文件；
4. 提供显示驱动用于调试和点亮显示屏；
5. 支持多线程。
6. 支持android recovery模式。

Little Kernel不仅代码量小,运行耗时少，更为重要的特点是代码可维护性好。

在Little Kernel构架中将处理器芯片归为平台，对于使用同一芯片的不同目标板则对应不同的目标。

2.1：LK的源码目录结构



App ———— 与应用有关的文件，包括正常启动模式，快速启动模式，恢复模式的代码，各种测试，控制台，写NAND Flash等代码。

Arch ———— 处理器体系结构有关的文件。主要支持ARM,X86构架。

Dev ———— 与设备有关的文件，这些设备包括帧缓冲控制，按键，有限网络，USB，电源管理模块等。

Include————各种头文件

Kernel————与LK系统相关，包括调试，延迟处理，事件，main，互斥，线程，定时器等。

Lib————各种库函数。

Make————编译相关的规则文件

Platform————平台相关的文件，支持的平台包括高通MSM系列，TI的OMAP系列，Telechips的TCC系列处理器。

Project————对应各种目标板的部分编译规则。

Scripts————脚本文件以及JTAG脚本。

Target————各种目标板对应的文件，包括引脚配置，时钟初始化，内存初始化，系统初始化，键盘初始化等。

3：LK的启动流程分析

1:从LK连接文件arch\arm\System-onesegment.ld可知，入口点在\_start。而\_start在汇编语言文件crt0.S中，从\_start跳转到reset，开始进行cpu的早期初始化，建立堆栈等常规的初始化任务，然后跳到kmain()函数执行。

2：进入kmain()函数后，首先执行thread\_init\_early()函数。LK支持多线程，因此需要该函数进行相关初始化，例如，初始化线程队列，初始化线程列表，创建运行线程等。

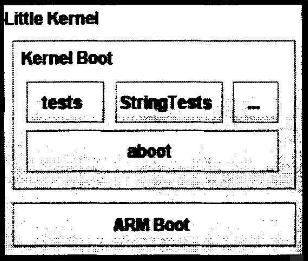
3：执行arch\_early\_init()函数。主要初始化MMU（内存管理单元），ARM NEON通用SIMD（单指令多数据）引擎。

4：执行platform\_early\_init()函数。lk将每种具体型号的ARM处理器归为一个平台。platform\_early\_init()函数对具体型号的处理器进行初始化，包括设置时钟，设置中断向量，初始化定时器/串口/I2C总线控制器等。

5：执行target\_early\_init(),call\_constructor(),heap\_init(),thread\_init()函数，而target\_early\_init()没有具体实现。

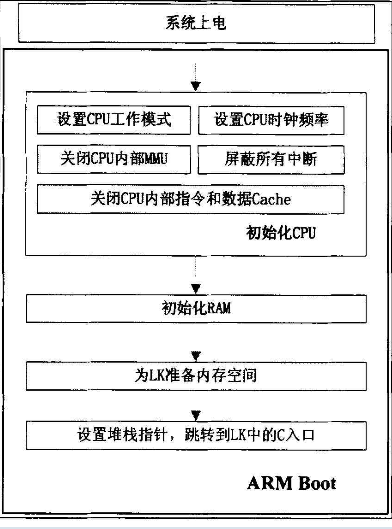
6：执行thread\_resume(thread\_create("bootstrap2", &bootstrap2, NULL, DEFAULT\_PRIORITY, DEFAULT\_STACK\_SIZE))，即进入引导第二阶段，对构架，平台，目标板进行具体初始化，并装载，引导内核程序。

LK采用传统的多阶段的Bootloader构架开发设计，分为ARMBoot模块和Kernel Boot模块俩大部分。



ARM Boot模块是系统上电后首先运行的程序，它的任务是为LK系统准备好必要的硬件环境和内存空间，并最终启动android引导代码。

一个典型的ARM启动代码一般是使用汇编语言编写，主要作用是初始化CPU和必备的硬件设备，比如DDR，中断控制器等。ARM Boot模块的工作流程：



1. 设置中断和异常向量

在片内RAM中，地址0x00-0x1C是32byte的中断向量表。每个中断占据4Byte存储空间。一旦有中断发生，ARM处理器便把pc指针强制置为表中所对应的中断类型的地址，是程序跳转到相应的中断处执行。当有多个中断发生时，处理器优先处理优先级高的中断。系统启动时，cpu从起始地址跳转到reset处开始执行：

\_start:

b reset //0x00:系统复位

b arm\_undefined //0x04: 未定义

b arm\_syscall //0x08:软件中断

b arm\_prefetch\_abort//0x0c:指令预取终止

b arm\_data\_abort //0x10：数据访问终止

b arm\_reserved//0x14:保留

b arm\_irq//0x18:普通中断

b arm\_fiq//0x1c:快速中断

2：屏蔽所有中断，禁止MMU,关闭cpu内部的指令cache和数据cache，为中断提供服务通常是操作系统设备驱动程序的责任，因此Bootloader的执行全过程无需响应任何中断。ARMBoot 阶段直接使用物理地址，无需MMU功能，另外，cpu内部的指令cache和数据也需要关闭：

/\* new thumb behavior, low exception vectors, i/d cache disable, mmu disabled \*/

mrc p15, 0, r0, c1, c0, 0

bic r0, r0, #(1<<15| 1<<13 | 1<<12)

bic r0, r0, #(1<<2 | 1<<0)

bic r0, r0, #(1<<1)

orr r0, r0, #(1<<5)

mcr p15, 0, r0, c1, c0, 0

3：配置系统所使用的寄存器

为Flash ,SRAM和DRAM等外设设置寄存器，并为他们分配地址空间，以确定其刷新频率，数据总线宽度等信息，初始化存储器系统。

4：为处理器的每个工作模式设置栈指针

ARM处理器的每个工作模式都需要设置单独的栈空间。处理器的各种工作模式都有自己通用的栈顶指针SP，通过将不同模式下的SP指向栈顶，可以完成堆栈的初始化。首先保存当前模式下的寄存器，并设置好堆栈：

mrs r0, cpsr

bic r0, r0, #0x1f

ldr r2, =abort\_stack\_top

接下来设置不同模式的SP指针指向栈顶：

orr r1, r0, #0x12 // irq

msr cpsr\_c, r1

ldr r13, =irq\_save\_spot

orr r1, r0, #0x11 // fiq

msr cpsr\_c, r1

mov sp, r2

orr r1, r0, #0x17 // abort

msr cpsr\_c, r1

mov sp, r2

orr r1, r0, #0x1b // undefined

msr cpsr\_c, r1

mov sp, r2

orr r1, r0, #0x1f // system

msr cpsr\_c, r1

mov sp, r2

orr r1, r0, #0x13 // supervisor

msr cpsr\_c, r1

mov sp, r2

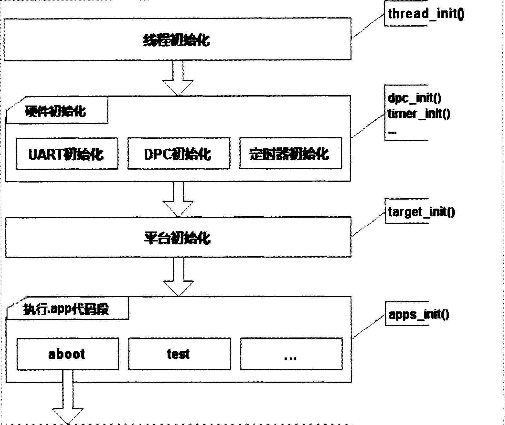
5：跳转到android boot入口：

到此为止，armboot阶段的硬件初始化工作基本完成，最后是将bios程序和androidboot程序拷贝到内存中，并跳转到LK入口函数：

bl kmain

b .

Kernel Boot 模块是LK系统的核心，从bootloader阶段设计角度来说，armboot阶段执行结束后，将android引导代码加载到内存中并开始执行kernel boot，属于bootloader的第二阶段。从功能设计角度来说，kernel boot模块不仅负责本阶段所需硬件设备的初始化，检测内存空间，加载系统内核和根文件系统，最终完成系统引导，同时承担着系统镜像的升级维护工作。Android系统支持基于USB的FALSH升级策略，主机与开发板之间的数据通信采用google定制的fasrboot协议，需要实现fastboot协议中定义的一系列主机命令，比如获取设备信息，擦除分区，烧写分区等，这些操作的具体实现依靠引导代码中的具体执行函数。Kernelboot模块的工作流程：





1. 线程初始化

LK系统中包括了许多应用程序，实现多应用同步执行的方法有很多，比如使用虚拟机，多线程和脚本等等。LK系统的开发采用了多线程技术，可以同步执行多种硬件设备的初始化以及在fastboot usb模式下同步检测按键中断功能，OEM厂商也可以使用该技术开发一些定制功能。

thread\_init();

2：硬件初始化

硬件初始化有DPC系统和定时器部分，串口的初始化。

3：平台初始化

本阶段最主要的工作是配置系统分区表，初始化flash和读取flash信息。

4：执行aboot程序

Aboot程序就是Kernel Boot 模块的核心。系统启动加载和镜像升级俩大主要功能都是在aboot程序中实现的，另外aboot程序还包括了硬件开机流程的处理。

在LK系统中，lk\app目录下的每个程序都是以多线程的形式运行的，每个程序都拥有一个app\_descriptor结构体，用来记录该程序的名称，初始化函数，入口函数和其他标识等信息，该结构体的定义：

struct app\_descriptor {

const char \*name;

app\_init init;

app\_entry entry;

unsigned int flags;

};

在每个程序中，使用APP\_START()和APP\_END宏来初始化app\_descriptor,这套宏命令指定将程序编译到.apps代码段，在app.h有这套宏的具体定义：

#define APP\_START(appname) struct app\_descriptor \_app\_##appname \_\_SECTION(".apps") = { .name = #appname,

#define APP\_END };

 kmain 源码分析：

**void kmain（）**

**{**

1.初始化进程（lk 中的简单进程）相关结构体。

**thread\_init\_early();**

2.做一些如 关闭 cache，使能 mmu 的 arm 相关工作。

**arch\_early\_init();**

  3.相关平台的早期初始化

**platform\_early\_init();**

 4.现在就一个函数跳转，初始化UART（板子相关）

**target\_early\_init();**

 5.构造函数相关初始化

**call\_constructors();**

6.lk系统相关的堆栈初始化

**heap\_init();**

    7.简短的初始化定时器对象

**thread\_init();**

8.lk系统控制器初始化（相关事件初始化）

**dpc\_init();**

9.初始化lk中的定时器

**timer\_init();**10.新建线程入口函数 bootstrap2 用于boot 工作 **thread\_resume(thread\_create("bootstrap2", &bootstrap2, NULL, DEFAULT\_PRIORITY, DEFAULT\_STACK\_SIZE));**

**}**

**以上与 boot 启动初始化相关函数是 arch\_early\_init、  platform\_early\_init 、bootstrap2，这些是启动的重点，我们下面慢慢来看。**

**arch\_early\_init()**

体系**[架构](http://lib.csdn.net/base/16" \o "大型网站架构知识库" \t "D:/Documents/My%20Knowledge/temp/_blank)**相关的初始化我们一般用的 ARM 体系

1.关闭cache

**arch\_disable\_cache(UCACHE);**

2.设置向量基地址（中断相关）

**set\_vector\_base(MEMBASE);**

3.初始化MMU

**arm\_mmu\_init();**

**4**.开启cache  

**arch\_enable\_cache(UCACHE)**

5.使能 cp10 和 cp11

**\_\_asm\_\_ volatile("mrc    p15, 0, %0, c1, c0, 2" : "=r" (val));**

**val |= (3<<22)|(3<<20);**

**\_\_asm\_\_ volatile("mcr    p15, 0, %0, c1, c0, 2" :: "r" (val));**

6.设置使能 fpexc 位 （中断相关）

**\_\_asm\_\_ volatile("mrc  p10, 7, %0, c8, c0, 0" : "=r" (val));**

**val |= (1<<30);**

**\_\_asm\_\_ volatile("mcr  p10, 7, %0, c8, c0, 0" :: "r" (val));**

7.使能循环计数寄存器

**\_\_asm\_\_ volatile("mrc    p15, 0, %0, c9, c12, 0" : "=r" (en));**

**en &= ~(1<<3);** /\*循环计算每个周期\*/

**en |= 1;**

**\_\_asm\_\_ volatile("mcr    p15, 0, %0, c9, c12, 0" :: "r" (en));**

 8.使能循环计数器

**en = (1<<31);  
       \_\_asm\_\_ volatile("mcr    p15, 0, %0, c9, c12, 1" :: "r" (en));**

**platform\_early\_init()**

board\_init();//平台板初始化

platform\_clock\_init();//时钟初始化

qgic\_init();//通用寄存器初始化

qtimer\_init();//获取定时器的频率

**bootstrap2**

         bootstrap2 在kmain的末尾以线程方式开启。主要分三步：platform\_init、target\_init、apps\_init。

**1.platform\_init**

              dprintf(INFO, "platform\_init()\n");

**2.target\_init**

              针对硬件平台进行设置。主要对分区表进行整合，初始化flash或mmc和读取分区信息

**3.apps\_init**

             apps\_init 是关键，对 LK 中所谓 app 初始化并运行起来，而 aboot\_init 就将在这里开始被运行，android linux 内核的加载工作就在 aboot\_init 中完成的 。

**aboot\_init**

**/\* call all the init routines \*/**

**for (app = &\_\_apps\_start; app != &\_\_apps\_end; app++) {**

**if (app->init)**

**app->init(app);**

**}**

**/\* start any that want to start on boot \*/**

**for (app = &\_\_apps\_start; app != &\_\_apps\_end; app++) {**

**if (app->entry && (app->flags & APP\_FLAG\_DONT\_START\_ON\_BOOT) == 0) {**

**start\_app(app);**

**}**

**Aboot.c:**

        1.设置NAND/ EMMC读取信息页面大小  
        **if (target\_is\_emmc\_boot())**

**{**

**page\_size = mmc\_page\_size();**

**page\_mask = page\_size - 1;**

**}**

**else**

**{**

**page\_size = flash\_page\_size();**

**page\_mask = page\_size - 1;**

**}**

**target\_display\_init(device.display\_panel);//LCD显示**

      2.读取按键信息，判断是正常开机，还是进入 fastboot ,还是进入recovery 模式

       。。。。。。。。。

**通过一系列的 if (keys\_get\_state() == XXX) 判断**

       。。。。。。。。。

3.normal\_boot:正常开机判断是emmc还是flash启动

**boot\_linux\_from\_flash**

**主要是内核的加载过程，我们的 boot.img 包含：kernel 头、kernel、ramdisk、second stage（可以没有）。**

1.读取boot 头部

\*\* +-----------------+

\*\* | boot header | 1 page

\*\* +-----------------+

\*\* | kernel | n pages

\*\* +-----------------+

\*\* | ramdisk | m pages

\*\* +-----------------+

\*\* | second stage | o pages

\*\* +-----------------+

\*\* | device tree | p pages

\*\* +-----------------+

ptable = flash\_get\_ptable();

ptn = ptable\_find(ptable, "boot");

**flash\_read(ptn, offset, buf, page\_size)**

2.读取 内核     **flash\_read(ptn, offset, (void \*)hdr->kernel\_addr, kernel\_actual)**

**offset += kernel\_actual;**

3.读取 ramdisk **flash\_read(ptn, offset, (void \*)hdr->ramdisk\_addr,ramdisk\_actual)**

**offset += ramdisk\_actual;**

1. **读取设备树**

**flash\_read(ptn, offset, (void \*) dt\_buf, page\_size)**

**flash\_read(ptn, offset, (void \*)table, dt\_hdr\_size)**

**flash\_read(ptn, offset + dt\_entry.offset,**

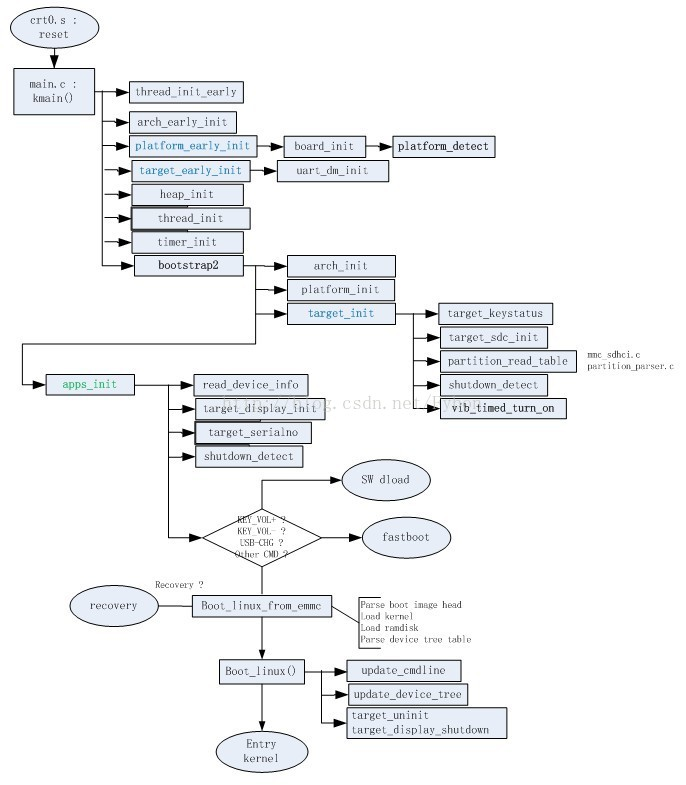
**(void \*)hdr->tags\_addr, dt\_entry.size)**

5.**启动内核**

**boot\_linux((void \*)hdr->kernel\_addr, (void \*)hdr->tags\_addr,**

**(const char \*)hdr->cmdline, board\_machtype(),**

**(void \*)hdr->ramdisk\_addr, hdr->ramdisk\_size);**//在boot\_linux 中entry(0,machtype,tags);从kernel加载在内核中的地址开始运行了。



Fastboot协议:

LK系统的另一个核心功能是基于Fastboot协议的镜像升级模式，Fastboot协议是pc主机通过USB与目标设备bootloader进行交互的一种通信协议，适用于linux,windows,osx等各种平台。

Fastboot协议通信是建立在usb协议之上的数据通信协议，首先，pc控制端与目标端必须建立usb连接物理通路，对于fastboot数据包的大小，根据usb协议版本的不同也有不同约定：在usb2.0协议标准中，规定了usb设备的三种传输速度，一般情况下，对于High-Speed usb(480Mb/s,usb2.0标准)，数据包的最大尺寸规定为512byte,对于FUll-Speed usb(12Mb/s,USB1.1标准)，数据包最大尺寸规定为64byte,fastboot不支持LOw-Speed usb。其次，fastboot协议的数据传输是由pc控制端驱动，目标设备booloader的工作是响应控制命令，执行相应的操作和反馈执行结果。并且数据传输过程相对目标端而言是完全同步的，这与多通道，双向，异步的ADB协议不同。

Fastboot协议的传输和组帧过程：

1. 主机发送命令，一个命令是一个ASCII字符串，并且只能包含在不大于64byte的数据包内。
2. 设备端使用一个不大于64byte的数据包做出相应。响应包的头4个字节是“OKAY”,“FAIL”,“DATA”或者“INFO”。响应包剩余字节可以包含ASCII格式的说明信息。4种响应包的含义如下：

A:INFO:剩余60个字节包含说明信息（提供进度报告或者诊断信息），这些说明信息应该由主机端控制显示。

B.FAIL:提示请求的命令失败。剩余60个字节可以提供一个文本形式的失败信息呈现给用户。通信终止。

C.DATA：请求的命令已经为数据传输阶段做好准备。一个DATA响应包是12byte。

1. 数据传输阶段。根据命令的不同，主机或设备端将发送指定大小的数据。比指定长度短的包总是可以接受的，零长度的包将被忽略。这一阶段会一直持续，直到设备端已经发送或接受了步骤2中DATA响应包指定的字节数为止。
2. 数据传输结束，设备端用一个不大于64byte的数据包响应。响应包的头4个字节是“OKAY”,“FAIL”或者“INFO”，类似于步骤2：

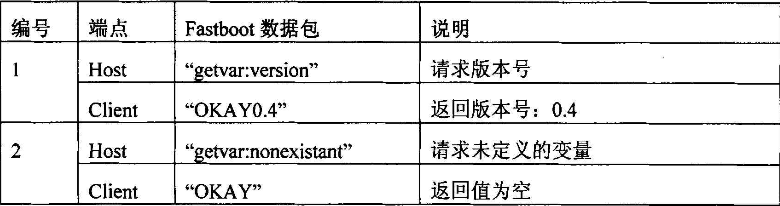
A:INFO:显示剩余60个字节，然后返回步骤4。

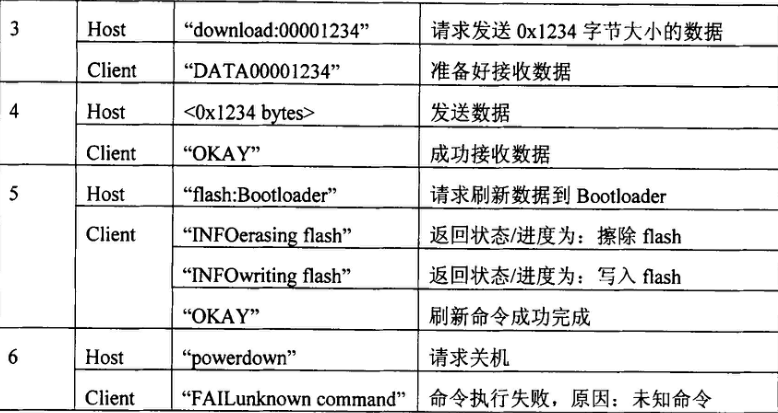
B:FAIL:显示剩余60个字节（如果存在）作为失败原因，命令失败，通信终止。

C:OKAY:命令执行成功。跳转到步骤5.

1. 命令执行完毕，结束通信

示列会话：





Fastboot USB模式是Google公司提出的一种基于USB设备的Android镜像升级方案，在该模式下，主机端通过fastboot控制程序与设备端的fastboot模块进行通信，实现设备信息查询，Flash分区插除，镜像烧写，设备重启等功能。主机端对设备端的控制通过flashboot命令来实现，数据包的格式必须遵守fastboot通信协议。LK系统fastboot模式实现了fastboot初始化，命令注册，设备端变量注册等，具体命令的执行流程需要在aboot程序中注册函数并且实现功能。

1. 模块初始化函数

Aboot程序在使用fastboot命令之前，必须先初始化fastboot模块。该函数在fastboot.h中头文件中的文件如下：

int fastboot\_init(void \*xfer\_buffer, unsigned max);

参数\*xfer\_buffer表示fastboot接受数据的缓存地址，max表示该缓存的大小。

Fastboot模块的注册流程如下：

1. 初始化UDC函数，初始化USB端点，分别需要一个输入端点（in）和一个输出端点（out）：

/\* initialize udc functions to use the default chipidea controller \*/

usb\_if.udc\_init = udc\_init;

usb\_if.udc\_register\_gadget = udc\_register\_gadget;

usb\_if.udc\_start = udc\_start;

usb\_if.udc\_stop = udc\_stop;

usb\_if.udc\_endpoint\_alloc = udc\_endpoint\_alloc;

usb\_if.udc\_request\_alloc = udc\_request\_alloc;

usb\_if.udc\_request\_free = udc\_request\_free;

usb\_if.usb\_read = hsusb\_usb\_read;

usb\_if.usb\_write = hsusb\_usb\_write;

输入端点：

in = usb\_if.udc\_endpoint\_alloc(UDC\_TYPE\_BULK\_IN, 512);

输出端点：

out = usb\_if.udc\_endpoint\_alloc(UDC\_TYPE\_BULK\_OUT, 512);

fastboot\_endpoints[0] = in;

fastboot\_endpoints[1] = out;

1. 初始化REQ请求包。USB设备使用REQ请求包来进行数据传输：

req = usb\_if.udc\_request\_alloc();

1. 注册设备端变量获取命令：

fastboot\_register("getvar:", cmd\_getvar);

1. 注册Download命令：

fastboot\_register("download:", cmd\_download);

1. 启动fastboot线程：

thr = thread\_create("fastboot", fastboot\_handler, 0, DEFAULT\_PRIORITY, 4096);

1. 命令注册函数

命令注册的功能是为每个主机端的命令注册执行函数句柄，主要通过fastboot\_register()函数实现。每个fastboot命令使用fastboot\_cmd结构体描述其详细信息：

struct fastboot\_cmd {

struct fastboot\_cmd \*next;

const char \*prefix;

unsigned prefix\_len;

void (\*handle)(const char \*arg, void \*data, unsigned sz);

};

命令注册的过程实际上为每个命令初始化fastboot\_cmd数据结构的过程。

1. 设备端变量注册函数

设备端变量是指关于设备和运行与设备之上的软件的各种信息。主机端使用指定命令(getvar:%s)来查询设备端的各种信息。Bootloader系统需要使用设备端变量注册函数为所以设备端变量赋值，以供主机端查询。

每个设备端变量使用fastboot\_var结构体来描述其详细信息，fastboot\_var定义如下：

struct fastboot\_var {

struct fastboot\_var \*next;

const char \*name;

const char \*value;

};

程序主要使用函数fastboot\_publish()实现设备端变量的注册。

Aboot程序在初始化阶段需要完成fastboot命令的注册，设备变量的注册以及fastboot模块的初始化。

命令函数注册：

struct fastboot\_cmd\_desc cmd\_list[] = {

{"", NULL},

{"flash:", cmd\_flash},

{"erase:", cmd\_erase},

{"boot", cmd\_boot},

{"continue", cmd\_continue},

{"reboot", cmd\_reboot},

{"reboot-bootloader", cmd\_reboot\_bootloader},

}

for (i = 1; i < fastboot\_cmds\_count; i++)

fastboot\_register(cmd\_list[i].name,cmd\_list[i].cb);

设备变量的注册：

fastboot\_publish("product", TARGET(BOARD));

fastboot\_publish("kernel", "lk");

fastboot\_publish("serialno", sn\_buf);

Gpio配置：

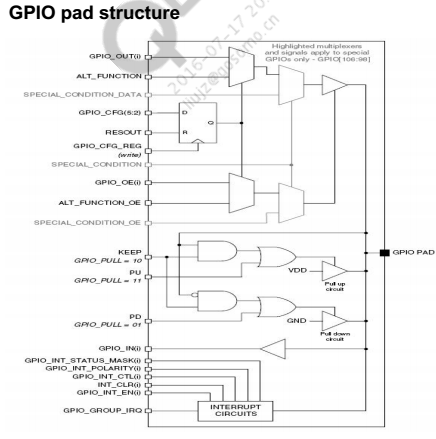
一：Gpio的分类：

1.gpio-msm8909.

2.gpio-pmic.

二：gpio-msm8909：

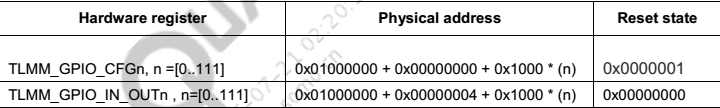
1. gpio的结构：

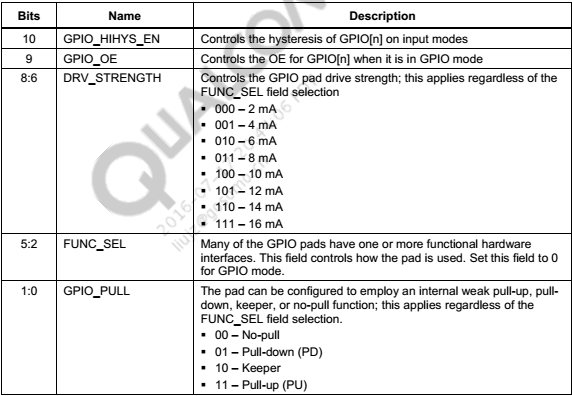


1. gpio的寄存器地址和配置：

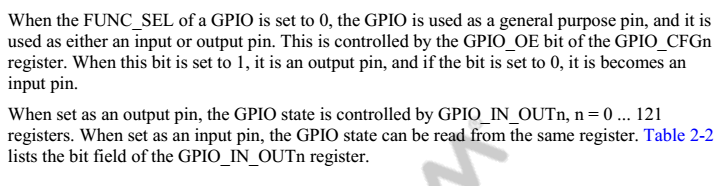
MSM8909 has 112 GPIOs

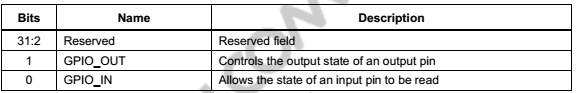
地址：





3：输入和输出配置

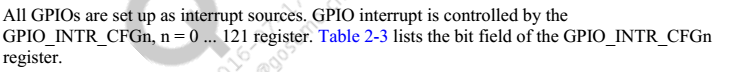


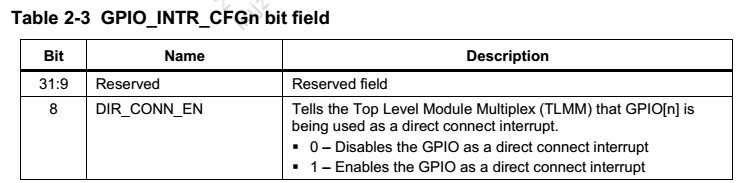


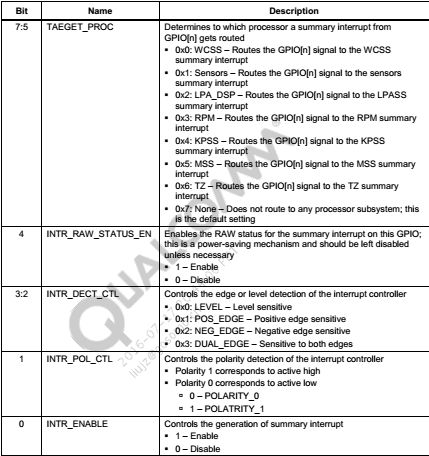
Gpio-msm.txt (kernel\documentation\devicetree\bindings\gpio)

Pinctrl-bindings.txt(kernel\documentation\devicetree\bindings\pinctrl)

1. 中断配置

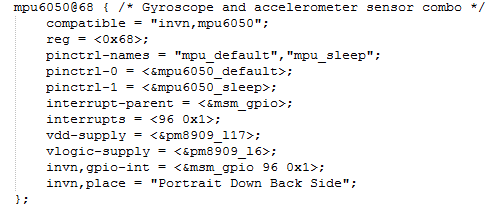


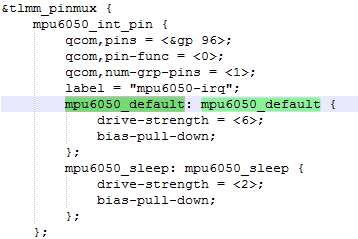




1. 引脚的三种kernel配置状态：

 Default  
 Active  
 Sleep

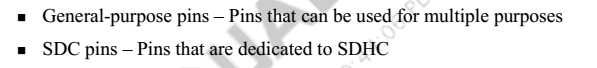


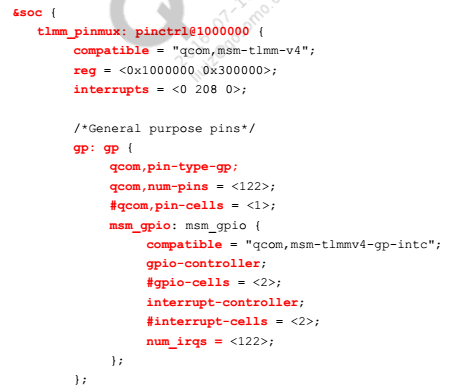


set\_state = pinctrl\_lookup\_state(pdev->dev->pins->p, “active”);

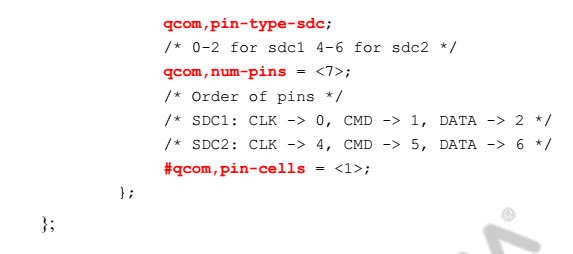
pinctrl\_select\_state(pdev->dev->pins->p, set\_state);

1. TLMM PIN类型：

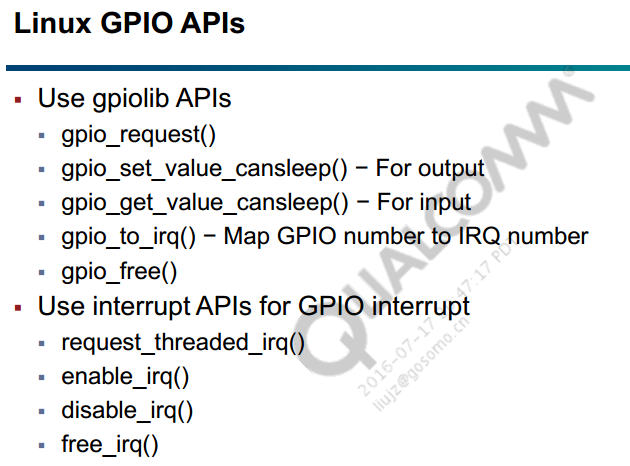








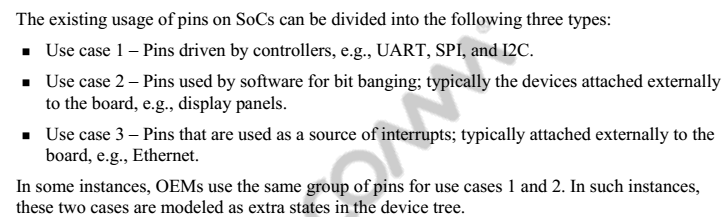
1. gpiolib api:



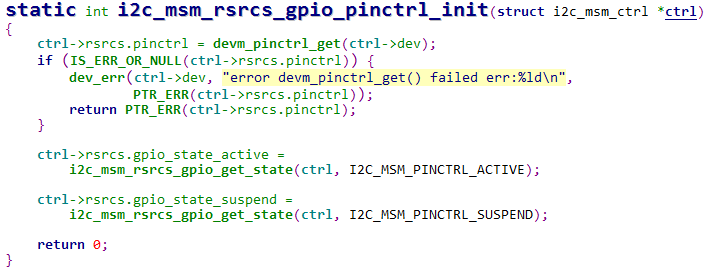
gpio\_direction\_in(client3\_data->reset\_gpio);

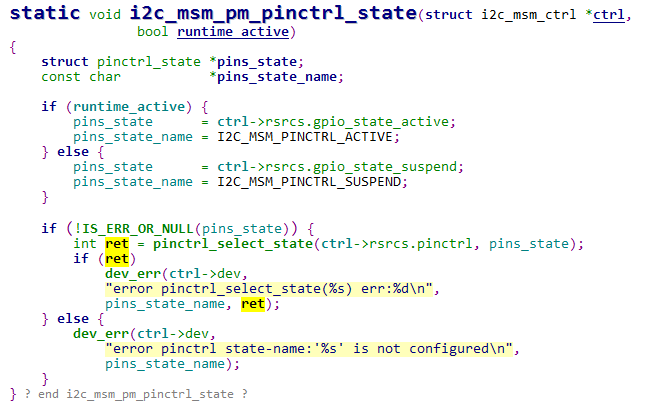
gpio\_direction\_out(client3\_data->reset\_gpio);  
gpio\_set\_value(client3\_data->reset\_gpio, 1);

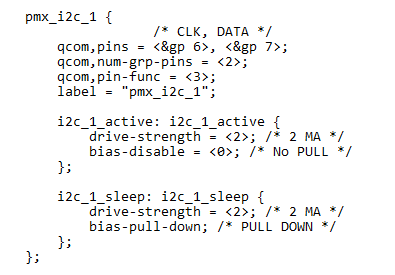
1. GPIO的三种使用情形：

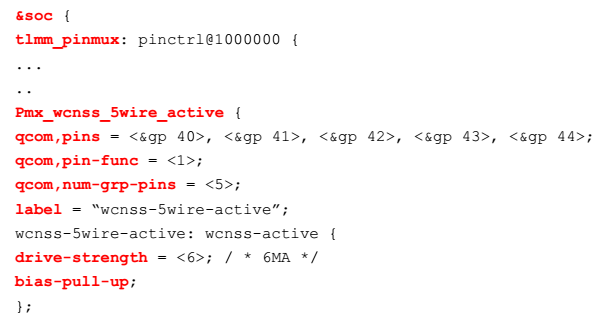


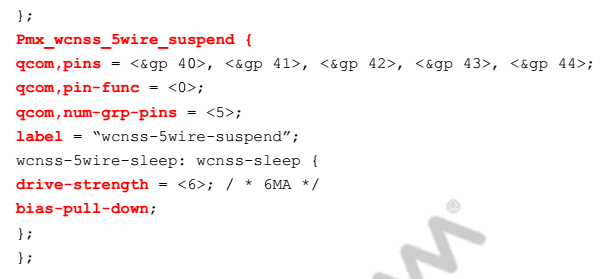
CASE 1:





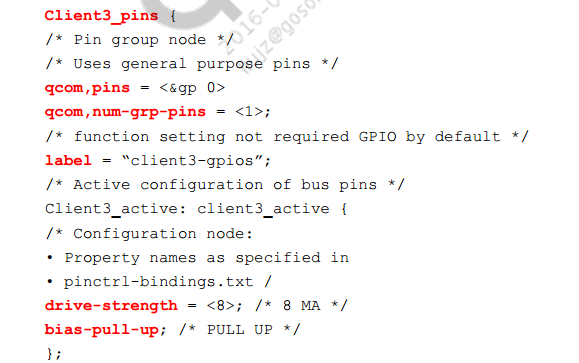


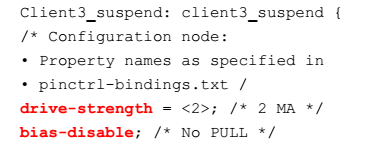






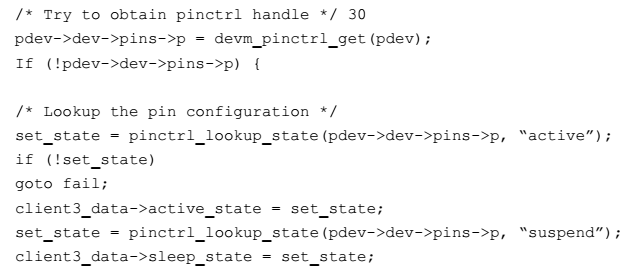
要声明为**gpio-controller**;



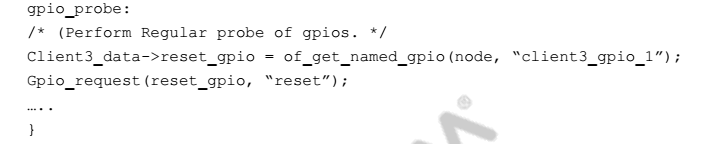


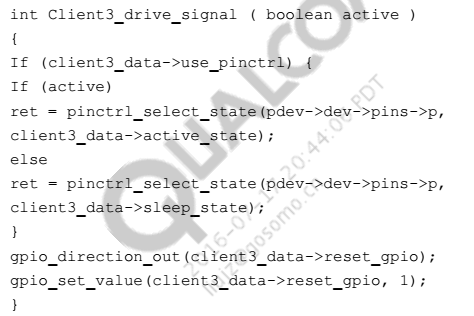
case 2 FUNC不变(0)

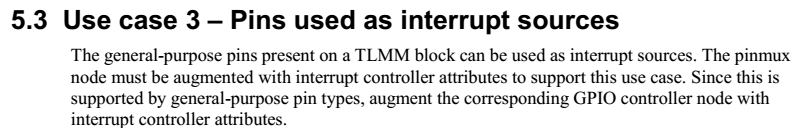
配置引脚功能：



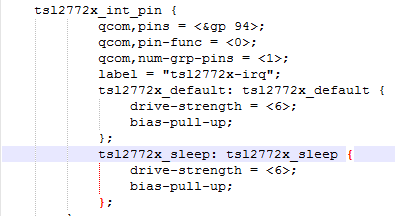
GPIO操作：

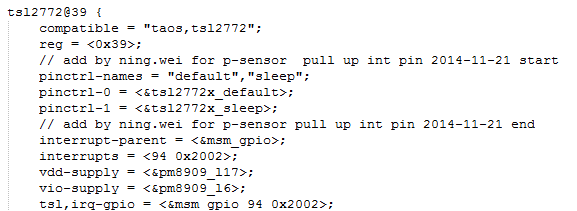






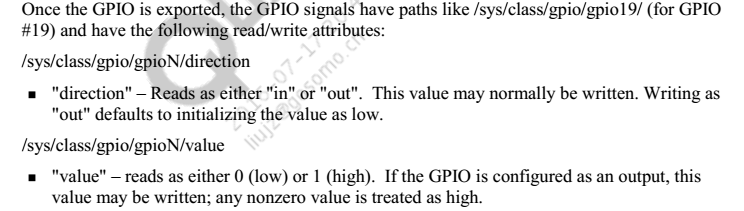
声明为**interrupt-controller**;

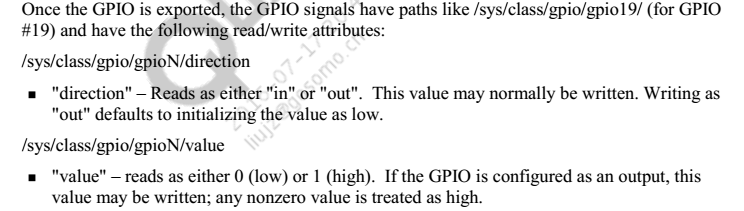




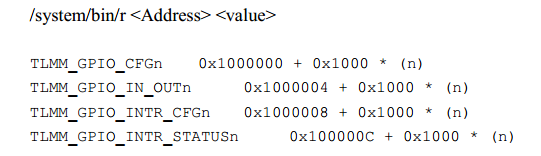
动态配置：gpio\_to\_irq

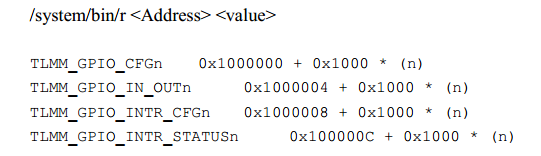
9：gpio调试



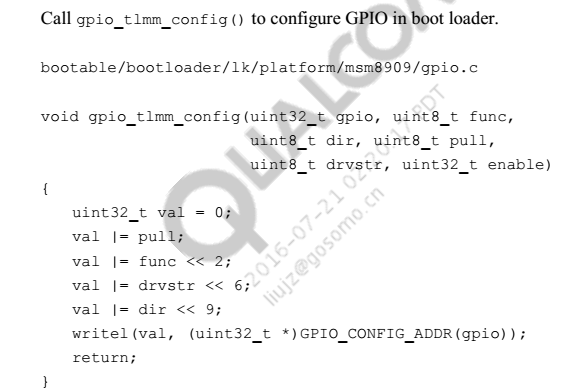


直接写寄存器：

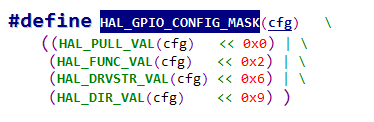




10:LK中的配置



sbl1配置：



qpnp-pin.txt

msm-pm8909.dtsi

msm8909-pm8909-mtp.dtsi