Q/LS

龙 芯 中 科 技 术 有 限 公 司 企 业 标 准

Q/LS 0018-2015

龙芯 CPU 开发系统 PMON 固件
开发规范

2015-03-01 发布　　2015-04-01 实施

龙 芯 中 科 技 术 有 限 公 司　　批 准

目 次

前　言

本规范是龙芯中科技术有限公司制定的企业规范，暂无国家相关行业通用规范可参考。

本规范涉及到龙芯 CPU 开发系统 PMON 固件开发的相关要求，内容包括 PMON 上电过程、内存初始化及地址窗口配置、驱动、中断路由、PMON 命令、固件与内核接口、编译配置、代码风格等方面。

本规范的起草单位：龙芯中科技术有限公司。

本规范的起草人：乔崇，成修治，王玉钱，马健，陈新科，黄沛，袁利，李雪峰，蒙天放，张宝祺。

本规范审核人：刘奇，邱吉，高翔，简方军，孟小甫，段玮，王焕东，苏孟豪，李文刚，符兴建，褚越杰。

本规范批准人：胡伟武。

龙芯 CPU 开发系统 PMON 固件开发规范

1 范围

本规范规定龙芯 2 号、3 号系列 CPU 开发系统 32 位 PMON 的详细开发要求,内容包括 PMON 上电过程、内存初始化及地址窗口配置、驱动及中断路由、PMON 命令、固件与内核接口、编译配置等方面。建议龙芯 1 号系列 CPU 开发 PMON 时参照此标准。建议其它系统厂商遵循此规范开发相关产品。

2 术语

a) 固件（Firmware):写入 ROM、EEPROM 等非易失存储器中的程序，负责控制和协调集成电路。

b) BIOS（Basic Input Output System)：基本输入输出系统，一组固化到主板的一个 ROM 芯片上的程序，它保存着计算机基本输入输出程序、系统设置信息、开机后自检程序和系统自启动程序。BIOS 与硬件系统集成在一起，也被称为固件，本规范中固件和 BIOS 不做区分。

c) HT（HyperTransport)：是一种为主板上的集成电路互连而设计的端到端总线技术，目的是加快芯片间的数据传输速度。HT 通常指 CPU 到主板芯片（或北桥）之间的连接总线，即 HT 总线。类似于 Intel 平台中的前端总线（FSB)，HT 按技术规格分有 HT1.0、HT2.0、HT3.0、HT3.1。

d) PCI（Peripheral Component Interconnect)：是连接电子计算机主板和外部设备的总线标准，用于定义局部总线的标准。

e) PMON：MIPS 架构机器上使用的一种具有 BIOS 部分功能的开放源码软件。

f) SMBIOS（System Management BIOS)：是主板或系统制造者以标准格式显示产品管理信息所需遵循的统一规范。DMI (Desktop Management Interface)是帮助收集电脑系统信息的管理系统，DMI 信息的收集必须在严格遵照 SMBIOS 规范的前提下进行。SMBIOS 和 DMI 是由行业指导机构 Desktop Management Task Force (DMTF)起草的开放性的技术标准。

g) XBAR: 龙芯 CPU 中用于片上地址路由的交叉开关， 二级 XBAR 中有 CPU 地址空间（包括 HT 空间）、DDR2 地址空间、以及 PCI 地址空间共三个 IP 相关的地址空间。

h) LS:是 Loongson 的缩写，通常作为芯片型号的前缀出现，如 LS3A 表示龙芯 3A 芯片。

i) LS2HSB：表示以龙芯 2H 作为南桥使用。

j) LS3A2H 系统：以龙芯 3A 为 CPU，龙芯 2H 为桥片的开发板系统。

k) LS3A780E 系统: 以龙芯 3A 为 CPU，AMD RS780E 为桥片的开发板系统。

l) LS3B780E 系统: 以龙芯 3B 为 CPU，AMD RS780E 为桥片的开发板系统。

m) LS1B 系统: 龙芯 1B SOC 的开发板系统。

n) LS2H 系统：龙芯 2H SOC 的开发板系统。

3 概述

本章涉及 PMON 上电启动过程、窗口配置、内存初始化过程等方面的内容。

3.1 上电启动过程

PMON 的上电启动过程如图 1 所示：

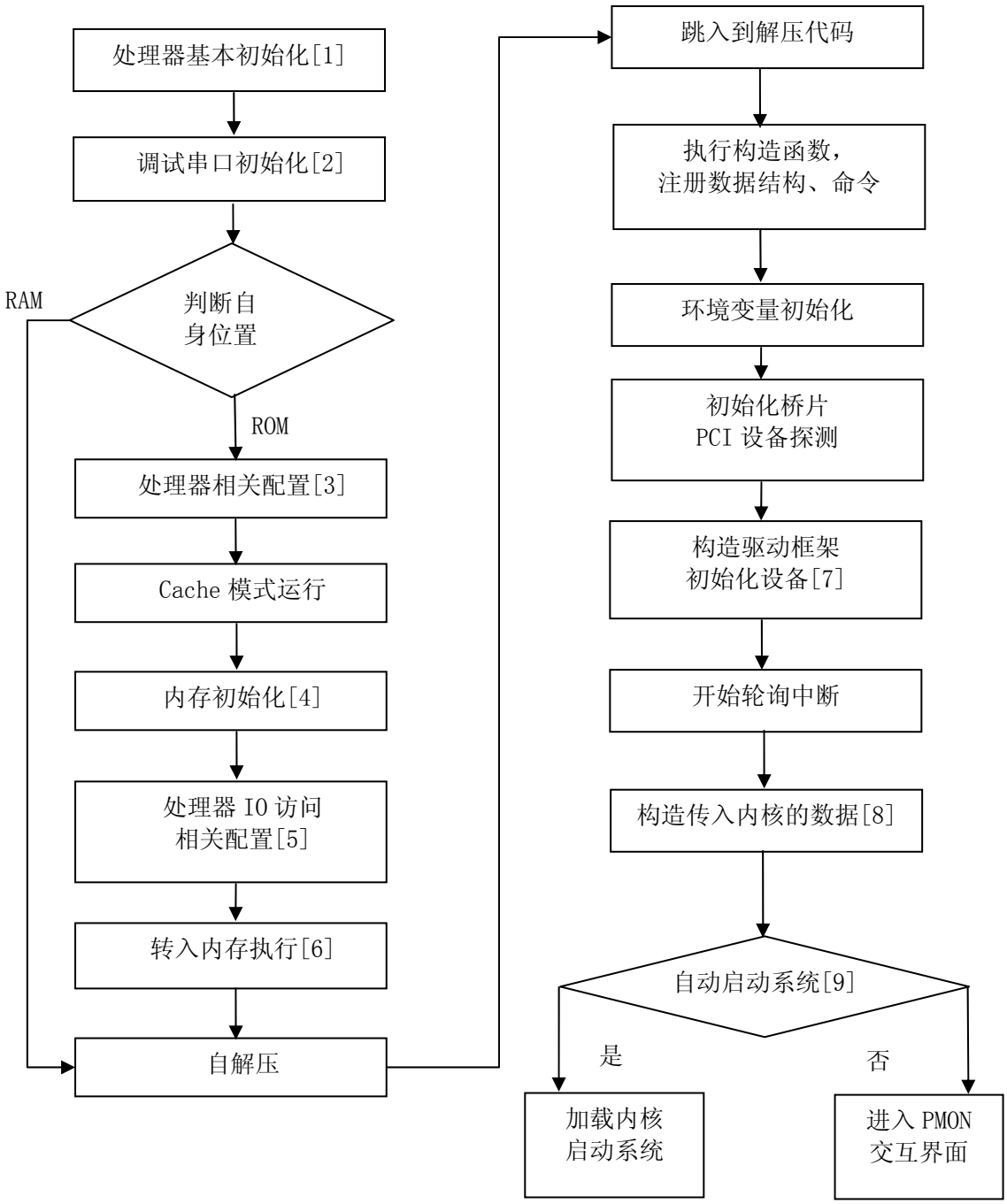


图 1 PMON 上电启动过程

针对启动过程的注释如下：

[1] PMON基本运行环境建立，即保证处理器正常运行的最基本设置，如：关中断，配置异常向量；如果多核，确定PMON启动核、从核自初始化Cache、TLB、清Mailbox（缓存寄存器）等。

[2] 如果串口在桥片上，则还需对桥片做初步配置。

[3] 处理器自身相关配置，如：修正频率、初始化Cache、CPU内部互联配置、非法地址处理。

[4] 如果连接内存的I2C控制器在桥片上，则还需对桥片做初步配置。

[5] 处理器级IO相关的一些配置，如：IO地址映射、桥片互联配置等。

[6] 内存运行的相关准备工作，如：拷贝代码到内存、设置堆栈、设置传参等。

[7] 初始化显示，以及PCI设备中断初始化。

[8] 构造提供给内核与系统的信息，如：内存布局、开发板类型等。

[9] 自启动系统时会查找boot.cfg或相关环境变量，根据查找结果来启动系统，如果找不到上述文件或变量，则会返回到命令行界面。

各类开发板系统详细的启动过程参见附录 E。

3.2 复位后 CPU 的初始状态

龙芯 CPU 上电启动后处理器核处于以下状态：

a） 小端模式（龙芯只支持小端模式）；

b） 特权等级处于内核态；

c） 浮点寄存器处于 32 位数据模式（PMON 启动过程中应将浮点寄存器配置为 64 位模式）；

d） 中断处于关闭态；

e） 非对齐访问会引发例外；

f） 64 位地址空间未使能，用户态 64 位操作未使能（龙芯 1 号系列除外）；

g） TLB 未初始化（PMON 启动过程中地址空间一直处于未映射段）；

h） 所有的 Cache 处于未初始化、未使能态(软件应该在初始化内存之前首先初始化并使能 Cache)。

3.3 窗口配置部分约定

3.3.1 窗口配置流程及注意事项

窗口的配置过程可分为以下步骤：

a） 配置用于阻挡猜测执行的地址窗口；

b） 配置不可预取的地址窗口，如 IO 寄存器映射地址区；

c） 配置高端内存及双通道交错窗口；

d） 配置 32 位地址到 64 位地址转换功能的窗口(GS464)；

e） 配置其它窗口，如龙芯 3A 中 HT 接收窗口等。

其中步骤 a）b）的配置需要在串口初始化完成且 TLB，Cache 尚未初始化之前进行；步骤 c）需要在内存容量探测及内存控制器初始化阶段完成；步骤 d）需要在用 32 位地址访问 IO 桥片之前完成，此项配置完成后即可使用 32 位的指针去访问 IO 桥片资源，如果不配置需要 64 位指针才能访问；步骤 e）需要在进入 C 语言前完成。

需要注意的是：

a） 窗口是有优先级的，序号越小，优先级越高；

b） 窗口配置的过程中，建议用汇编语言或内嵌汇编语言对其进行配置；

c） 有些窗口是有默认功能的，在配置之前最好先保留通过此窗口的通路；

d） 有些窗口寄存器是只写的，配置完后不一定能读出写入的值；

e） 各窗口相关的寄存器描述及窗口的作用请查阅各芯片的用户手册。

地址窗口与 CPU 的具体设计密切相关，一般由龙芯公司统一进行配置和发布。

3.3.2 窗口配置约定及示例

在窗口配置过程中，定义如下的宏对两级交叉开关进行配置:

```
#define set_Xbar_win(xbarbase，mas_i, win_i, base, mask, mmap) \
  li  t0, xbarbase ;\
  daddiu t0, t0, mas_i*0x100 ;\
  daddiu t0, t0, win_i*0x8 ;\
  dli t1, base ;\
  sd  t1, 0x0(t0) ;\
  dli t1, mask ;\
  sd  t1, 0x40(t0) ;\
  dli t1, mmap ;\
  sd  t1, 0x80(t0)
```

对上述约定说明如下：

a） 需要用到寄存器:t0，t1。

b） 参数及其含义：

xbarbase：交叉开关配置寄存器的基址；

mas_i：主设备号；

win_i：此主设备的窗口号（即第win_i个窗口）；

base：接收地址基址；

mask：接收地址掩码；

mmap：转换后的地址基址及路由的次设备号。

c） 功能：检查输入的地址是否满足过滤条件(in_addr&mask)==base ，若满足此条件则将进来的地址 in_addr 转换后发出去，转换规则是(in_addr&(~mask))|mmap ；反之则不做处理，直接向下转发。C 代码可实现为：

(in_addr&mask)==base?out_addr=(in_addr&(~mask))|mmap:out_addr=in_addr。

例如在LS3A2H系统中，对南桥LS2HSB的PCIE控制器(在XBAR2上的主设备上)窗口的配置如下：

```
#PCIE window
#define LS2HSB_XBAR2_BASE 0xbbd80000
#define LS2HSB_PCIE_MASTER 4
set_Xbar_win(LS2HSB_XBAR2_BASE,        LS2HSB_PCIE_MASTER,        0,
0x0000000000000000, 0xffffffffc0000000, 0x00000010800000f3) # 0~1G
set_Xbar_win(LS2HSB_XBAR2_BASE,        LS2HSB_PCIE_MASTER,        1,
0x0000000040000000, 0xffffffffc0000000, 0x00000020800000f3) # 1~2G
set_Xbar_win(LS2HSB_XBAR2_BASE,        LS2HSB_PCIE_MASTER,        2,
0x0000000080000000, 0xffffffffc0000000, 0x00000030800000f3) # 2~3G
set_Xbar_win(LS2HSB_XBAR2_BASE,        LS2HSB_PCIE_MASTER,        3,
0x00000000c0000000, 0xffffffffc0000000, 0x00000040800000f3) # 3~4G
set_Xbar_win(LS2HSB_XBAR2_BASE,4,              LS2HSB_PCIE_MASTER,
0xffffffff80000000, 0xffffffffc0000000, 0x00000030800000f3) # 2~3G
set_Xbar_win(LS2HSB_XBAR2_BASE,4,              LS2HSB_PCIE_MASTER,
0xffffffffc0000000, 0xffffffffc0000000, 0x00000040800000f3) # 3~4G
```

例如在LS3A2H系统中，对南桥LS2HSB的IODMA控制器（在XBAR1上的主设备上）窗口的配置如下：

```
#define LS2HSB_XBAR1_BASE 0xbbd82000
set_Xbar_win(LS2HSB_XBAR1_BASE , 6 , 7, 0x0000000000000000,
0x0000000000000000, 0x00000000000000f0) # others, all to L2$ （默认通路为第二个窗口，因此在使用窗口 2 之前应先将此通路备用）
set_Xbar_win (LS2HSB_XBAR1_BASE , 6 , 0, 0x0000001080000000,
0xffffffff0000000, 0x00000000000000f6)
set_Xbar_win (LS2HSB_XBAR1_BASE , 6 , 1, 0x0000001080000000,
0xffffffffc0000000, 0x00000001000000f6)
set_Xbar_win (LS2HSB_XBAR1_BASE , 6 , 2, 0x0000002080000000,
0xffffffffc0000000, 0x00000001400000f6)
set_Xbar_win (LS2HSB_XBAR1_BASE , 6 , 3, 0x0000003080000000,
0xffffffffc0000000, 0x00000001800000f6)
set_Xbar_win (LS2HSB_XBAR1_BASE , 6 , 4, 0x0000004080000000,
0xffffffffc0000000, 0x00000001c00000f6)
```

可用如下方法检验配置结果：

```
PMON> d8 0xffffffffbbd82600 30
ffffffffbbd82600: 0000001080000000 0000001080000000 ................
ffffffffbbd82610: 0000002080000000 0000003080000000 .... .......0...
ffffffffbbd82620: 0000004080000000 000000f080000000 ....@...........
ffffffffbbd82630: 0000000040000000 0000000000000000 ...@............
ffffffffbbd82640: ffffffff0000000 fffffffc0000000 ................
ffffffffbbd82650: fffffffc0000000 fffffffc0000000 ................
ffffffffbbd82660: fffffffc0000000 fffffffff000000 ................
ffffffffbbd82670: fffffffc0000000 0000000000000000 ................
ffffffffbbd82680: 00000000000000f6 00000001000000f6 ................
ffffffffbbd82690: 00000001400000f6 00000001800000f6 ...@............
ffffffffbbd826a0: 00000001c00000f6 000000fdf80000f6 ................
ffffffffbbd826b0: 00000000000000f0 00000000000000f0 ................
ffffffffbbd826c0: 0000000000000000 0000000000000000 ................
ffffffffbbd826d0: 0000000000000000 0000000000000000 ................
ffffffffbbd826e0: 0000000000000000 0000000000000000 ................
```

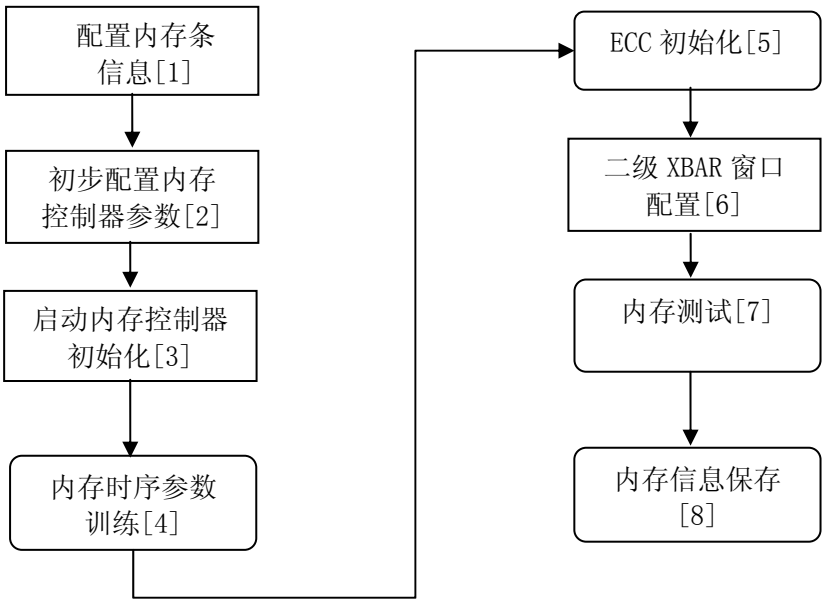
通过上述的配置，连接在LS2HSB PCIE控制器上的设备作为主设备发起DMA访问时，发出的总线地址范围是0x00000000至0xFFFFFFFF，会依次被LS2HSB二级交叉开关、LS2HSB一级交叉开关、LS2HSB IODMA模块、HT接收窗口、龙芯3A一级交叉开关、龙芯3A二级交叉开关转换最终到达内存控制器访存。

附录 D 给出了 LS3A2H 窗口配置的具体实现。

3.4 内存初始化

内存初始化的主要工作包括：配置内存控制器、配置二级XBAR。

目前PMON内存初始化的过程如下图所示：



注：1. 方角矩形框为必选步骤，圆角矩形框为可选步骤。

2. 内存时序参数训练，也称内存训练（代码中命名为 RB_level）。

图 2　PMON 内存初始化流程图

[1] 配置内存条信息

该步骤的作用是获得需要的内存信息，包括：内存类型（DDR2 还是 DDR3）、内存条类型（UDIMM、RDIMM、是否带 ECC）、内存条的数据线宽度、内存颗粒的 Bank 个数/地址线的行数/列数、是否进行地址 mirror、与 CPU 的引脚连接方式（CS 的映射方式）、内存的容量等。分为两种模式（通过配置选项 AUTO_DDR_CONFIG 选择）：根据内存条 SPD 自动检测和人工配置。通过 SPD 自动检测时，需要提供内存条 SPD 所在的 I2C 总线地址。

[2] 初步配置内存控制器参数

该过程分为三步：

a) 根据内存类型从 BIOS 数据段装载基本参数

根据内存类型（DDR3 或 DDR2，UDIMM（包括 SODIMM） 或 RDIMM，以及是否进行了内存训练）从 BIOS 数据段的不同位置选择相应的基本参数。当尚未进行内存训练时，根据 DDR3/2 以及 UDIMM/RDIMM 共 4 种组合选择一个位置加载；当进行了内存训练时，直接从训练后保存的内存参数位置加载。

b) 根据内存条信息重新配置部分参数

重新配置 1 中所述除了内存类型和内存条类型以外的其他内存条信息。如果加载的是训练后保存的参数，则跳过此步骤。

c) 单独修改部分参数

调试时使用。

[3] 启动内存控制器初始化

向内存控制器的参数 param_start 写入值 1，使得硬件开始内存控制器的初始化过程，软件轮询初

始化是否结束。

[4] 内存时序参数训练

调用 ARB_level 函数，对部分时序参数进行重新配置。

[5] ECC 初始化

对于使能 ECC 的情况，进行 ECC 初始化。

[6] 二级 XBAR 窗口配置

根据使能的内存控制器个数和每个控制器的内存容量来配置二级 XBAR。

[7] 内存测试

测试内存读写的正确性，同时可以测试二级 XBAR 窗口配置是否正确。供调试使用。

[8] 内存信息保存

保存内存频率、内存条的 ID 等相关信息，供智能的内存训练使用。内存训练可以根据内存频率、内存条是否发生过改变来决定是否需要重新训练。

配置内存控制器的过程为步骤[1]—[5]。当系统中有多个内存控制器时，内存控制器需要分别进行配置（重复步骤[1]—[5]）。

配置选项 ARB_LEVEL 决定是否包含步骤：[4]、[8]。宏定义 DEBUG_DDR 决定是否包含步骤[7]。宏定义 DISABLE_DIMM_ECC 决定是否包含步骤[5]。

如果使能内存训练，则配置内存控制器需要在 Cache 初始化并使能之后进行。

内存初始化的代码封装在 loongson*_ddr[2]_config.S 中，使用 64 位的寄存器 s1 进行参数传递，使用时需要正确设置输入参数（s1）。内存初始化完成后会设置寄存器 msize（寄存器 s2）的值，告诉后续代码系统的内存容量，每个节点对应 8 位，从低到高分别对应节点 0/1/2/3，单位为 512MB。

4 PMON 的系统需求

4.1 PMON 的地址空间分配

PMON 在完成内存控制器的配置前，执行的代码位于 NOR FLASH 中，完成内存控制器的配置后，将位于 NOR FLASH 中的 gzip 压缩的映像文件拷贝到内存，然后进行解压缩操作，最后，处理器跳转到解压缩后的 PMON 入口 initmips 开始执行；

PMON 将物理地址 0x0F000000～0x0FFFFFFF 之间的内存用作保留空间，操作系统内核不得使用该段内存；其中 0x0F400000～0x0F7FFFFF 用作运行时的堆空间，0x0F000000～0x0F3FFFFF 用作 PMON 的代码段与数据段；

参见《龙芯 CPU 开发系统固件与内核接口详细规范》的 4.2 节“地址空间表”。

4.2 PMON 低 256M 的空间分配

0x0F000000～0x0FFFFFFF 空间的具体划分如下表：

表 1 PMON 低 256M 空间分配表

地址	描述
0x0F80 0000 - 0x0FFF FFFF	固件与内核接口参数地址，在操作系统启动后此空间要保留给固件使用，其中 0x0FFFE000 是 SMBIOS 的基地址，0x0FFF0000-0x0FFF0110，用来保存内存条信息，供智能的内存训练使用
0x0F40 0000 - 0x0F7F 0000	PMON 堆栈
0x0F00 0000 – 0x0F3F 0000	PMON 代码段和数据段

0x0020 0000 - 0x0EFF FFFF	内核和 RamDisk 加载
0x0000 0000 – 0x001F FFFF	保留

4.3 PMON 的板卡存储需求

板卡需要最小 512KB 的 FLASH 芯片，建议使用 1MB 的 FLASH 芯片。

5 PMON 启动及内核加载

龙芯系列 CPU 启动地址都固定为 0xBFC00000。支持从以下两种介质启动，但针对具体 CPU 型号或硬件设计，可能在同一设计不会同时支持，具体按实际情况而定。

a) 直接寻址介质：NOR FLASH

现有龙芯 CPU 家族都支持此启动方式，也是目前设计中最常见的方式。总线接口依据具体 CPU 型号有所差别，其支持总线类型的有 SPI、LPC、LocalBus。此空间大小为 1MB。

b) 间接寻址介质：NAND FLASH

某些型号的 CPU 支持从 NAND FLASH 或者其它存储介质启动，此时介质起始的 1K 大小会被加载到芯片内部的 RAM 中，映射到 0xBFC00000 或者其它位置。此 1K 大小的程序/数据可以被处理器直接寻址，支持 32 位访问。

下表为 PMON 支持加载内核的存储介质：

表 2 PMON 支持加载内核的存储介质列表

存储介质	接口	文件系统/协议
DVD-ROM/CD-ROM	SATA/IDE/USB	ISO 9660
机械/固态硬盘	SATA/IDE	fat32/ext2/ext3/ext4
RAID 磁盘阵列	RAID 0~5	fat32/ext2/ext3/ext4
U 盘/移动硬盘	USB	fat32/ext2/ext3/ext4
网络节点	网络	tftp，http
NAND FLASH/NOR FLASH	LocalBus/相应控制器	无（直接物理块读/写）

6 驱动与设备路由

6.1 PMON 设备驱动模型

通过 pmoncfg 命令来自动生成 Makefile 和.h 文件 pmoncfg，代码位于 tools/pmoncfg.

pmoncfg 命令格式如下：

```
pmoncfg configfile
```

执行这个命令，pmoncfg 读 configfile 从里面查找包含下面关键字的行

```
machine target arch sysarch
```

然后包含如下配置文件 ：

Targets/<target>/conf/files.<target>

Targets/<sysarch>/conf/files.<sysarch>

conf/files

在这3个配置文件中还可能通过include命令包含其他配置文件。在配置文件中包含设备总线描述，例如下面代码是 Targets/ls1b/conf/ls1b里面的描述：

```
define  mainbus {}
device  mainbus: sys
attach  mainbus at root
file    pmon/dev/mainbus.c  mainbus  needs-flag
```

在make cfg调用pmoncfg后生成ioconf.c文件，该文件位于Targets/<主板型号>/compile/目录下。

在生成的文件中，cfdata 是一个设备树，pv 数组定义一个设备的父设备。每个节点的父设备都是一个数组，在设备的 cfdata 结构中定义数组的开始。 为了简化，PMON 将所有设备的设备的父设备数组放在一个数值 pv 中. 其中-1 表示数组的结束.

以上述龙芯 1B 中的 cfdata[1]为例子，cfdata[1].pv == pv+2，pv[2]==7，说明其父设备为 cfdata[7]。

设备自动初始化也是利用树进行递归的初始化，其代码实现在 configure 函数中，从 mainbus 开始依次初始化各级设备。其中的 cfdata[1]有两个关键数据结构。例如对于 USB 设备而言是 usb_ca 和 usb_cd，其具体定义在 usb_storage.c 中，usb_storage.c 定义的驱动结构如下：

```
struct cfattach usb_ca = {
 .ca_devsize = sizeof(struct device),
 .ca_match  = usb_match,
 .ca_attach  = usb_attach,
};

struct cfdriver usb_cd = {
 .cd_devs = NULL,
 .cd_name = "usb",
 .cd_class = DV_DISK,
};
```

cfattach，cfdriver 是每个驱动必须定义的结构体，分别指向函数列表和设备表述。

configure函数在每个平台的tgt_machdep.c中的tgt_devconfig函数中调用。

在 configure 函数中，config_rootfound 首先调用 config_rootsearch 找到设备并返回 cf 结构，然后调用 config_attach 分配设备结构，并挂接设备。

config_attach 除了调用设备驱动的 attach 函数外，还调用 TAILQ_INSERT_TAIL(&alldevs, dev, dv_list)将设备挂在设备列表里面。 可以通过 devls 命令列出。

PMON 代码从 BSD 代码移植过来，设备驱动用文件来访问，普通设置用 open()访问，网络设备用 socket()来访问。通过 open()访问 PMON 下的设备时，格式如下表：

表 3 open()访问 PMON 文件例子

格式	说明
/dev/fs/ext2@wd0/boot/vmlinux	ext2/ext3 wd0 硬盘下 boot/vmlinux 文件
/dev/fs/fat@usb0/boot/vmlinux	fat usb0 U 盘下 boot/vmlinux 文件
/dev/disk/wd0	整个硬盘 wd0
/dev/tty1	tty1 设备
tftp://10.0.0.1/vmlinux	tftp 10.0.0.1 上的 vmlinux 文件

6.2 PMON 中断路由

龙芯开发系统中断路由分为可配置和不可配置两类，对于中断路由不可配置的部分参照芯片相关手册说明进行配置，对于中断可以路由的桥片（目前主要是以 AMD 的 780E 为桥片的方案），规范如下。

下面以 LS3A780E 来说明 PCIE 的中断号在 PMON 分配过程。写入设备的 interupt_line 寄存器，该代码实现在 sb700_interrupt_fixup 中。中断设置依据以下几点:

a) 北桥 PCIE 的 bus0 上每个设备的中断 A-D 到内部 PCI INTA-INTD 的路由如下表：

表 4 北桥 PCIE 中断路由

设备号	设备位设置
2	INTA->INTC
3	INTA->INTD
4	INTA->INTA
5	INTA->INTB
6	INTA->INTC
7	INTA->INTD
9	INTA->INTB
10	INTA->INTC

b) 南桥 USB 等设备到内部 PCI INITA-G，中断路由是固定的；

c) 0xc00，0xc01 设置内部 PCI INITA-D 到 8259 后的中断号路由如下表：

表 5 0xc00，0xc01 寄存器定义

寄存器	域名	位	默认值	描述
0xc00	Pci_Intr_Index	7:0	00h	PCI 中断索引，选择哪个 PCI 中断进行映射
				0h:INTA#
				1h:INTB#
				2h:INTC#
				3h:INTD#
				4h:Interrupt generated by ACPI#
				5h:Interrupt generated by Sm Bus#
				6h:Interrupt generated by ACPI#
				7h:Interrupt generated by ACPI#
				8h:Interrupt generated by ACPI#
				9h:INTE#
				Ah:INTF#
				Bh:INTG#
				Ch:INTH#
0xc01	Pci_Intr_Data	7:0	00h	PCI 重映射寄存器，根据 Pci_Intr_Index 到 PIC irq 进行 PCI 中断地址映射

d) 0x4d0，0x4d1 设置中断是电平触发还是边沿触发，PCIE 中断要求电平触发，LPC 上设备中断如串口要求边沿触发。因此 LPC 中断不能和 PCIE 分配相同的中断号；

表 6 0x4d0 寄存器定义

域名	位	默认值	描述
IRQ0Control	0	0b	1:Level, 0:Edge
IRQ1Control	1	0b	1:Level, 0:Edge
保留	2	0b	1:Level, 0:Edge
IRQ3Control	3	0b	1:Level, 0:Edge
IRQ4Control	4	0b	1:Level, 0:Edge
IRQ5Control	5	0b	1:Level, 0:Edge
IRQ6Control	6	0b	1:Level, 0:Edge
IRQ7Control	7	0b	1:Level, 0:Edge
IRQ8Control	8	0b	(只读) Always:Edge
IRQ9Control	9	0b	1:Level, 0:Edge
IRQ10Control	10	0b	1:Level, 0:Edge
IRQ11Control	11	0b	1:Level, 0:Edge
IRQ12Control	12	0b	1:Level, 0:Edge
IRQ13Control	13	0b	1:Level, 0:Edge
IRQ14Control	14	0b	1:Level, 0:Edge
IRQ15Control	15	0b	1:Level, 0:Edge

e) PCI MSI/MSIX 中断到 HT 的中断映射。LS3A780E 支持 MSI 中断，MSI/MSIX 中断的寄存器地址是 0xFEE00000。PCIE MSI/MSIX 中断消息写到 0xFEE00000 的数据会转换成 HT 中断发送给龙芯 3A。MSI/MSIX 中断号分配在内核中由软件灵活配置。

LS3A780E 的 PMON 中断分配需要调用 sb700_interrupt_fixup 函数，如下所示。其中 sb700_interrupt_fixup0 主要设置南桥上设备等集成中断。sb700_interrupt_fixup1 自动扫描 PCI 设备根据前面提到中断分配规则，自动分配中断。sb700_interrupt_fixup 代码片段如下：

```
void sb700_interrupt_fixup(void)
{
 sb700_interrupt_fixup0();
 sb700_interrupt_fixup1();
}
```

7 固件与内核接口

本章主要描述与内核接口的两个头文件 smbios.h 及 bootparam.h 及相关的说明。

7.1 与内核接口的约定

1) 与内核接口的数据结构的头文件参见附录 A，结构的具体说明参见《龙芯 CPU 开发系统固件与内核接口详细规范》。

头文件的命名约定为 bootparam.h，相应实现函数的 c 文件命名约定为 bootparam.c，以上两个文件放在 pmon/common 目录。

要求在固件中初始化好所有的结构体或服务，最终封装到 boot_params 的结构体中，将该结构体指针 bp 赋值给 a2 寄存器。在 main.c 中调用 boot_params 的指针，并将指针赋值给 a2 寄存器。具体实现是在 pmon/common/main.c 中的 initstack 函数中调用 md_setargs 实现，在 machdep.c 中该函数定义如下:

```
md_setargs(struct trapframe *tf, register_t a1, register_t a2,
     register_t a3, register_t a4)
{
  if (tf == NULL)
     tf = cpuinfotab[whatcpu];
  tf->a0 = a1;
  tf->a1 = a2;
  tf->a2 = a3;
  tf->a3 = a4;
}
```

2) SMBIOS

SMBIOS 要求实现目前建议必须实现的 SMBIOS 类别如下: 兼容 SMBIOS 2.3 规范版本， 必须实现包含以下 8 个数据表结构：

a) BIOS 信息(Type 0)

b) 系统信息 (Type 1)

c) 产品信息 (Type 2)

d) 处理器信息 (Type 4)

e) 物理存储阵列 (Type 16)

f) 存储设备(Type 17)

g) 温度传感器 (Type 28)

h) 表格结束指示 (Type 127)

具体可参见《龙芯 CPU 开发系统固件与内核接口详细规范》。

SMBIOS 要求放在 pmon/common 目录，SMBIOS 的结构入口地址定义为：

```
#define SMBIOS_PHYSICAL_ADDRESS 0x8fffe000
```

具体实现是在 pmon/common/smbios/smbios.h 中定义。

7.2 运行时服务的约定

如下函数约定在 PMON 中实现

1） init_reset_system()

概述：init_reset_system 用于复位整个平台，包括处理器、设备、以及重启系统。

函数原形 void init_reset_system(struct efi_reset_system_t *reset)

参数:reset

相关的定义：

```
struct efi_reset_system_t {
u64 ResetCold;
u64 ResetWarm;
u64 ResetType;
u64 Shutdown;
u64 DoSuspend; /* NULL if not support */
};
```

表 7 efi_reset_system_t 结构含义说明

名称	描述
ResetCold	冷启动， 将全系统电路恢复到初始状态
ResetWarm	热启动， CPU 设置为初始状态，其他正常
ResetType	为后续预留
Shutdown	使系统进入类似 ACPI G2/S5(OS 不会保存和回复系统上下文，仅有很少设备如键盘，鼠标供电)或 G3(关机)状态
DoSuspend	为后续预留

返回的状态码: 该函数没有返回码。

目前在 PMON 里必须实现 Shutdown 和 ResetWarm 两个函数，对应的实现函数名称统一为 poweroff_kernel 和 reboot_kernel

8 PMON 的人机界面

本章含命令的格式说明、命令相关的数据结构及实现约定、命令分类，命令索引和简表参见附录 C。

8.1 命令格式

PMON 命令采用类 UNIX 的命令格式：

command [options] [arguments]

command：命令

options: --单词 或 -单字

argument:参数(文件名、路径或其他)

在查看命令帮助 h 时，会出现[]，<>，|等符号，它们的含义如下：

[] 表示是可选的；

<> 表示是可变化的；

x|y|z 表示只能选择一个；

-abc 表示三个参数(或任何二个)的混合使用

8.2 命令相关的数据结构

1） 命令选项描述数据结构 Optdesc

```
typedef struct Optdesc {
    const char    *name;
    const char    *desc;
} Optdesc;
```

表 8 Optdesc 数据结构说明

名称	描述
name	命令选项名
desc	命令选项描述

此数据结构描述了命令选项的描述，在帮助命令 help 时会显示的这些命令选项信息，此结构以结构数组形式出现，在定义时，以{0}表示结尾项。

2） 命令数据结构 Cmd

```
typedef struct Cmd {
const char    *name;
const char    *opts;
const Optdesc  *optdesc;
const char    *desc;
int          (*func) __P((int, char *[]));
int          minac;
int          maxac;
int          flag;
#define  CMD_REPEAT   1  /* Command is repeatable */
#define  CMD_HIDE  2  /* Command is hidden */
#define  CMD_ALIAS 4  /* Alias for another command name */
} Cmd;
```

成员说明:

表 9 Cmd 数据结构说明

名称	描述
name	命令名
opts	该命令可选的参数，对命令的解释，命令执行的函数，最少参数个数，最多参数个数，该命令的类型（参见flag说明）
optdesc	可选参数的解释
desc	命令描述
(*func) __P((int, char *[]))	命令执行的函数

minac	最小参数个数
maxac	最大参数个数
flag	这个命令的类型，如 CMD_REPEAT(Command is repeatable)或 CMD_HIDE(Command is hidden)或其他命令的别名 CMD_ALIAS(Alias for another command name)

3) 命令注册数组定义：

命令表注册结构数组约定如下:

```
static const Cmd Cmds[] ={
        {命令所属组名},
        {命令相关定义},
        …
        {0,0}
}
```

Cmds 数组 Cmds[]最大约定为 100，以{0，0}表示结尾。

在定义这个数组时，第一项是命令所属组名，目前的组名约定如下：

表 10 命令在 Cmds 中所属组名约定

名称	描述
Debugger	调试相关命令，如命令g、r等
Boot and Load	启动或加载，如命令load等
Memory	内存显示/设置相关的，如命令 m等
Pci	PCI相关的命令
Misc	杂项，如命令 log devls等
Network	网络相关，如ping
Environment	环境相关的，如命令set
Shell	Shell命令相关的，如命令sh、vers等

4) 命令修改目录约定：

公用的命令应放在目录 pmon/cmd 目录下，自己添加的通用命令放在 cmd/loongson 中，某开发板特殊的命令放在各自的 Targets/<boardname>下，例如 LS3A780E 的特殊命令放在 Targets/Bonito3a780e/Bonito 下。

8.3 命令的相关的数据结构及函数实现

以 load 命令为例来说明命令的实现，代码见 pmon/cmds/load.c，关键函数及数据结构说明如下表：

表 11 load 命令相关的关键函数说明

名称	描述
const Optdesc cmd_load_opts[]	定义命令选项的名称及描述，help 命令时会输出，以{0}表示结束
static int nload (int argc, char **argv)	命令的具体执行函数
static const Cmd Cmds[]	命令定义数组，第一项是命令所属组名，最后以{0,0}表结束
static void init_cmd()	命令的注册函数

1) 定义命令选项及描述数组

```
const Optdesc        cmd_nload_opts[] =
{
        {"-s", "don't clear old symbols"},
        {"-b", "don't clear breakpoints"},
         … …
```

```
        {"path", "path and filename"},
        {0}
    };
```

2）命令执行函数:

```
    int cmd_nload (argc, argv)
    int argc; char **argv;
    {
    int ret;
    ret = spawn ("load", nload, argc, argv);
    return (ret & ~0xff) ? 1 : (signed char)ret;
    }
```

3）命令注册表数组

```
    static const Cmd Cmds[] =
    {
    {"Boot and Load"}, //命令所属组名
    {"load","[-beastifr][-o offs]",
    cmd_nload_opts,
    "load file",
    cmd_nload, 1, 16, 0}, // 命令定义
    {0, 0} //结束定义
    };
```

4）命令注册函数

```
    static void init_cmd __P((void)) __attribute__ ((constructor));
    static void init_cmd() {
        cmdlist_expand(Cmds, 1);
     }
```

这里 init_cmd 函数被声明为__attribute__((constructor)) ，表示在 main 函数被调用之前调用，而使用__P((void))是 C 语言的__P 语法，是为了 ANSI C 和非 ANSI C 编译器都可兼容编译设置。

数据结构 struct Cmd 在 include/pmon.h 文件中， 命令相关的函数都在 pmon/cmds/的文件中定义，cmdlist_expand()函数在 pmon/cmds/cmdtables.c 中定义。

命令的解析是在/pmon/common/cmdparser.c 中的 do_cmp(p)函数里实现的。

8.4 命令分类

表 12 命令分类

功能	指令列表
下载文件或镜像	加载文件 (load) 启动 (boot) 网络启动 (netboot) 磁盘启动 (scsiboot) 定义符号 (sym) 列出符号 (ls) 列出磁盘分区 (fdisk) 加载 RamDisk 镜像 (initrd)

功能	指令列表
	列出设备（devls）
显示/设置寄存器 显示/设置内存	显示/设置寄存器（r） 修改内存（m，m1,m2,m4,m8） 显示内存（d,d1,d2,d4,d8） 反汇编内存（l） 填充内存（fill） 拷贝内存（copy，memcpy64） 搜索内存（search） 导出内存（dump） 比较内存（compare，mycmp） PCI 设备相关（pcs，pciscan，pcicfg）
执行控制， 断点	开始执行（g） 显示/设置断点（b） 删除断点（db） 单步执行（t / to） 打印出当前的函数调用栈（bt） 继续执行（c） 执行子程序（call）
网络及通讯	网卡配置（ifconfig，ifaddr） 串口协议（xmodem，ymodem）
杂项和环境控制	帮助（h） 关于 PMON（about） 进入 DBX 模式（debug） 显示历史信息（hi） 显示/设置环境变量（set/eset，env） 设置终端参数（stty） 日期设置/日期显示（date） 写/擦除 FLASH 内存区（flash） 透明模式（tr） 刷 Cache（flush） 显示版本号（ver） 命令 Shell（sh） 分页（more） 重启 PMON（reboot） PMON 睡眠（sleep） 循环执行命令（loop） 为磁盘设备赋值（losetup） 关机（poweroff） 计算数值（eval）
诊断	内存测试（mt,mtest） 地址空间漏洞扫描（mscan） 网络 PING 测试（ping）

9 PMON 编译、配置、目录结构及代码风格

9.1 编译器的版本

PMON 的升级交叉工具链有两种，一种是 Gcc2.95，另一种是 Gcc4.4.0。2014 年 6 月 17 日之前的 PMON 代码使用的工具链来自基于 Gcc-2.95 和 Binutils-2.11 的工具链。该工具链在编译 C 代码和汇编代码时偶尔会将立即数加载指令序列（lui/addiu 或者 lui/daddiu）编错（变为 daddiu/addiu），从而导致生成的 PMON 无法正常启动。此后开始使用 Gcc-4.4.0 交叉工具链。

Gcc-4.4.0 与 Gcc-2.95 的主要区别：

1) Gcc-2.95 对 64bit 下的 O32 ABI 支持与 Gcc-4.4.0 不一致，Gcc-2.95 直接支持 64bit 的整形变量（这与 O32 ABI 不一致），即对于如下的`int func(unsigned long long A0， unsigned long long A1)， Gcc-2.95 使用寄存器 a0 和 a1 传参；而 Gcc-4.4.0 则使用 Gcc-4.4.0 则使用寄存

器 a0，a1，a2，a3（分别对应 A0 的低 32 位与高 32 位，A1 的低 32bit 和高 32bit）与 O32 ABI 一致；

2) Gcc-4.4.0 使用更加严格的语法检查，包括但不限于：

a) 函数定义与函数原型声明不一致；

b) 函数的多重定义；

c) 条件编译中#ifndef 使用常数而非宏判断条件编译。

9.2 PMON 编译配置

目前，PMON 可以在编译时进行配置，PMON 的配置项主要分为两部分:一是对命令的配置，另一种是对模块的配置。

1) 配置项语法有如下两种：

option <**DEFINE**>，用于增加宏定义，例如option BONITOEL 将在生成的Makefile 中定义 BONITOEL。

select <**MODULE**>，选择需要编译的模块，例如select mod_framebuffer。

2) 对命令和模块的配置

关于对命令或模块的裁剪，其主要是通过修改配置文件，将不需要的命令注释掉即可，具体的配置文件一般包括:

```
Targets/<target>/conf/files.<target>
Targets/<sysarch>/conf/files.<sysarch>
conf/files
```

这三个配置文件还通过 include 包含其他配置文件。

9.3 PMON 的目录结构

PMON 主要的目录结构参见附录 B 说明,新增文件须参照附录 B 中的说明将文件添加到相应的目录中。

9.4 PMON 代码风格

PMON 主要采用 C 语言写成，部分文件采用汇编，其 C 语言部分编码风格参见附录 F 的建议。

附录 A

（规范性附录）

内核与固件接口头文件

bootparam.h

```
#ifndef __ASM_MACH_LOONGSON_BOOT_PARAM_H_
#define __ASM_MACH_LOONGSON_BOOT_PARAM_H_

#define SYSTEM_RAM_LOW   1
#define SYSTEM_RAM_HIGH  2
#define MEM_RESERVED     3
#define PCI_IO           4
#define PCI_MEM          5
#define LOONGSON_CFG_REG 6
#define VIDEO_ROM        7
#define ADAPTER_ROM      8
#define ACPI_TABLE       9
#define SMBIOS_TABLE     10
#define UMA_VIDEO_RAM    11
#define MAX_MEMORY_TYPE  12
#define LOONGSON3_BOOT_MEM_MAP_MAX 128
struct efi_memory_map_loongson {
  u16 vers; /* version of efi_memory_map */
  u32 nr_map;  /* number of memory_maps */
  u32 mem_freq; /* memory frequence */
  struct mem_map {
     u32 node_id;  /* node_id which memory attached to */
     u32 mem_type; /* system memory, pci memory, pci io, etc. */
     u64 mem_start;   /* memory map start address */
     u32 mem_size; /* each memory_map size, not the total size */
  } map[LOONGSON3_BOOT_MEM_MAP_MAX];
} __attribute__((packed));

enum loongson_cpu_type {
  Loongson_2E = 0,
  Loongson_2F = 1,
  Loongson_3A = 2,
  Loongson_3B = 3,
  Loongson_1A = 4,
```

```
 35   Loongson_1B = 5
 36  };
 37
 38  /*
 39   * Capability and feature descriptor structure for MIPS CPU
 40   */
 41  struct efi_cpuinfo_loongson {
 42   u16 vers;    /* version of efi_cpuinfo_loongson */
 43   u32 processor_id; /* PRID, e.g. 6305, 6306 */
 44   u32 cputype;  /* Loongson_3A/3B, etc. */
 45   u32 total_node;   /* num of total numa nodes */
 46   u32 cpu_startup_core_id; /* Core id */
 47     u16 reserved_cores_mask; /*Reserved CPU Core mask*/
 48   u32 cpu_clock_freq; /* cpu_clock */
 49   u32 nr_cpus;
 50  }__attribute__((packed));

 51  #define MAX_UARTS 64
 52  struct uart_device {
 53   u32 iotype; /* see include/linux/serial_core.h */
 54   u32 uartclk;
 55   u32 int_offset;
 56   u64 uart_base;
 57  } __attribute__((packed));
 58
 59  #define MAX_SENSORS 64
 60  #define SENSOR_TEMPER  0x00000001
 61  #define SENSOR_VOLTAGE 0x00000002
 62  #define SENSOR_FAN    0x00000004
 63  struct sensor_device {
 64   char name[32];  /* a formal name */
 65   char label[64]; /* a flexible description */
 66   u32 type;      /* SENSOR_* */
 67   u32 id;        /* instance id of a sensor-class */
 68   u32               fan_policy;              /*              see
arch/mips/include/asm/mach-loongson/loongson_hwmon.h */
 69   u32 fan_percent;/* only for constant speed policy */
 70   u64 base_addr;  /* base address of device registers */
 71  } __attribute__((packed));
```

```
72
73 struct system_loongson {
74   u16 vers;    /* version of system_loongson */
75   u32 ccnuma_smp; /* 0: no numa; 1: has numa */
76   u32 sing_double_channel; /* 1:single; 2:double */
77   u32 nr_uarts;
78   struct uart_device uarts[MAX_UARTS];
79   u32 nr_sensors;
80   struct sensor_device sensors[MAX_SENSORS];
81   char has_ec;
82   char ec_name[32];
83   u64 ec_base_addr;
84   char has_tcm;
85   char tcm_name[32];
86   u64 tcm_base_addr;
87   u64 workarounds; /* see workarounds.h */
88 } __attribute__((packed));
89
90 struct irq_source_routing_table {
91   u16 vers;
92   u16 size;
93   u16 rtr_bus;
94   u16 rtr_devfn;
95   u32 vendor;
96   u32 device;
97   u32 PIC_type;  /* conform use HT or PCI to route to CPU-PIC */
98   u64 ht_int_bit; /* 3A: 1<<24; 3B: 1<<16 */
99   u64 ht_enable;  /* irqs used in this PIC */
100  u32 node_id;   /* node id: 0x0-0; 0x1-1; 0x10-2; 0x11-3 */
101  u64 pci_mem_start_addr;
102  u64 pci_mem_end_addr;
103  u64 pci_io_start_addr;
104  u64 pci_io_end_addr;
105  u64 pci_config_addr;
106  u32 dma_mask_bits;
107 } __attribute__((packed));
108
109 struct interface_info {
110  u16 vers; /* version of the specificition */
```

```
111  u16 size;
112  u8  flag;
113  char description[64];
114 } __attribute__((packed));
115
116 #define MAX_RESOURCE_NUMBER 128
117 struct resource_loongson {
118  u64 start; /* resource start address */
119  u64 end;   /* resource end address */
120  char name[64];
121  u32 flags;
122 };
123
124 struct archdev_data {};  /* arch specific additions */
125
126 struct board_devices {
127  char name[64];    /* hold the device name */
128  u32 num_resources; /* number of device_resource */
129  /* for each device's resource */
130  struct resource_loongson resource[MAX_RESOURCE_NUMBER];
131  /* arch specific additions */
132  struct archdev_data archdata;
133 };
134
135 struct loongson_special_attribute {
136  u16 vers;    /* version of this special */
137  char special_name[64]; /* special_atribute_name */
138  u32 loongson_special_type; /* type of special device */
139  /* for each device's resource */
140  struct resource_loongson resource[MAX_RESOURCE_NUMBER];
141 };
142
143 struct loongson_params {
144  u64 memory_offset;  /* efi_memory_map_loongson struct offset */
145  u64 cpu_offset;     /* efi_cpuinfo_loongson struct offset */
146  u64 system_offset;  /* system_loongson struct offset */
147  u64 irq_offset;     /* irq_source_routing_table struct offset */
148  u64 interface_offset;   /* interface_info struct offset */
149  u64 special_offset; /* loongson_special_attribute struct offset
```

```
*/
 150  u64 boarddev_table_offset;  /* board_devices offset */
 151 };
 152
 153 struct smbios_tables {
 154  u16 vers;    /* version of smbios */
 155  u64 vga_bios; /* vga_bios address */
 156  struct loongson_params lp;
 157 };
 158
 159 struct efi_reset_system_t {
 160  u64 ResetCold;
 161  u64 ResetWarm;
 162  u64 ResetType;
 163  u64 Shutdown;
 164  u64 DoSuspend; /* NULL if not support */
 165 };
 166
 167 struct efi_loongson {
 168  u64 mps;  /* MPS table */
 169  u64 acpi; /* ACPI table (IA64 ext 0.71) */
 170  u64 acpi20;  /* ACPI table (ACPI 2.0) */
 171  struct smbios_tables smbios;  /* SM BIOS table */
 172  u64 sal_systab;  /* SAL system table */
 173  u64 boot_info;   /* boot info table */
 174 };
 175
 176 struct boot_params {
 177  struct efi_loongson efi;
 178  struct efi_reset_system_t reset_system;
 179 };
 180
 181 struct loongson_system_configuration {
 182  u32 nr_cpus;
 183  u32 nr_nodes;
 184  int cores_per_node;
 185  int cores_per_package;
 186  enum loongson_cpu_type cputype;
 187  u64 ht_control_base;
```

```
188  u64 pci_mem_start_addr;
189  u64 pci_mem_end_addr;
190  u64 pci_io_base;
191  u64 restart_addr;
192  u64 poweroff_addr;
193  u64 suspend_addr;
194  u64 vgabios_addr;
195  u32 dma_mask_bits;
196 };
197
198 extern struct efi_memory_map_loongson *loongson_memmap;
199 extern struct loongson_system_configuration loongson_sysconf;
200
201 extern u32 loongson_nr_uarts;
202 extern struct uart_device loongson_uarts[MAX_UARTS];
203 extern char loongson_ecname[32];
204 extern u32 loongson_nr_sensors;
205 extern struct sensor_device loongson_sensors[MAX_SENSORS];
206 #endif
```

附录 B
（规范性附录）
PMON 目录结构

B.1　PMON 目录结构见表 13，重要文件说明见表 14。

表 13 PMON 目录结构

目录	子目录	说明
conf	conf/	公共部分配置文件目录
lib	lib/libc	库文件目录
	lib/libc/arch/mips	
	lib/libz	
	lib/libz/arch/mips	
	lib/libm	
	lib/libm/arch/mips	
doc		文档目录
examples	examples/	示例程序目录
pmon	pmon/arch/mips	与 MIPS 相关的目录
	pmon/arch/mips/mm	与内存配置相关的目录
	pmon/cmds	各种命令目录
	pmon/cmds/cmd_main	main 菜单相关目录
	pmon/cmds/gzip	gzip 压缩解压相关目录
	pmon/cmds/lwdhcp	DHCP 相关目录
	pmon/cmds/test	测试程序
	pmon/common	公用的文件目录
	pmon/common/smbios	SMBIOS 相关目录
	pmon/dev	与设备相关的目录
	pmon/fs	文件系统目录
	pmon/fs/cramfs	cramfs 文件系统目录
	pmon/fs/cramfs/include	cramfs 文件系统头文件目录
	pmon/fs/yaffs2	yaffs2 文件系统目录
	pmon/loaders	loader 相关目录
	pmon/loaders/zmodem	Zmodem 协议目录
	pmon/netio	网络相关目录
fb	fb/	显示相关目录
sys	sys/arch/mips/include	MIPS 体系头文件目录
	sys/dev	各种设备目录
	sys/dev/ata	ATA 设备目录
	sys/dev/gmac	GMAC 目录
	sys/dev/ic	各种 IC 目录
	sys/dev/mii	网卡相关
	sys/dev/pci	PCI 目录
	sys/dev/usb	USB 目录
	sys/dev/fd	软驱相关
	sys/dev/nand	NAND FLASH 目录
	sys/kern	PMON 的内核
	sys/linux	Linux 相关头文件
	sys/net	网络核心代码
	sys/netinet	各种网络协议目录
	sys/scsi	SCSI 设备目录
	sys/sys/	头文件
	sys/vm	内存管理

include	include	头文件
tools	tools	各种工具目录
Targets		各种开发板相关目录
x86emu		模拟执行显卡 BIOS 中的 X86 指令代码目录

表 14 重要文件说明

文件	说明
pmon/common/main.c	main 函数相关文件
conf/files	配置文件
Targets/<Boardname>/conf/Bonito.<Boardname>	与板卡相关的配置文件
Targets/<Boardname>/conf/files.Bonito<Boardname>	与板卡相关的配置文件
Targets/<Boardname>/Bonito/start.S	板卡相关的启动文件汇编部分
Targets/<Boardname>/Bonito/tgt_machdep.c	板卡相关的启动文件 c 部分
pmon/fs/devfs.c	块设备读写相关文件
pmon/arch/mips/cache.S	Cache 相关文件
pmon/cmds/bootparam.c	向内核传参相关文件
pmon/dev/kbd.h	PS2 键盘相关文件
pmon/dev/kbd.c	
sys/dev/usb/usb_storage.c	USB 存储类设备相关文件
sys/dev/usb/usb-ohci.c	OHCI 相关文件
sys/dev/gmac/gmac_plat.c	GMAC 相关文件
sys/dev/gmac/gmac_network_interface.h	
sys/dev/gmac/if_gmac.c	
sys/dev/gmac/gmac_dev.h	
sys/dev/gmac/gmac_network_interface.c	
sys/dev/gmac/gmac_host.h	
sys/dev/gmac/gmac_dev.c	
sys/dev/pci/ahci.c	AHCI 相关文件
sys/dev/pci/ahcisata.c	SATA 设备相关文件
sys/dev/pci/pciconf.c	PCI 总线相关文件
pmon/dev/ns16550.h	串口相关文件
pmon/dev/ns16550.c	
pmon/cmds/cmd_main/cmd_main.h	main 命令相关文件
pmon/cmds/cmd_main/cmd_main.c	
pmon/cmds/cmd_main/window.c	

附录 C

（规范性附录）

命令索引及简表

C.1　命令索引见表 15。

表 15 PMON 命令索引

命令	功能	执行控制	显示或修改	环境	杂项	诊断
about	关于 PMON				Y	
b	断点	Y				
boot	启动命令		Y			
bt	堆栈回溯		Y			
c	继续执行	Y				
call	调用	Y				
compare	对比两块内存		Y			
copy	拷贝		Y			
d(d1,d2,d4,d8)	显示		Y			
date	显示/设置日期时间			Y		
db	删除断点	Y				
debug	调试				Y	
devls	列出当前所有设备		Y			
dump	通过 RS232/Ethernet 上载		Y			
env	察看环境变量			Y		
eset	编辑变量			Y		
eval	计算算式的数值				Y	
fdisk	显示磁盘分区		Y			
fill	填写		Y			
flash	写或擦除 FLASH 区			Y		
flush	刷数据或指令 Cache				Y	
g	执行	Y				
h	帮助				Y	
hi	历史			Y		
ifaddr	配置网卡 IP 地址			Y		
ifconfig	网卡配置			Y		
initrd	加载 RamDisk 镜像		Y			
l	列表（反汇编）		Y			
load	通过 RS232/Ethernet 下载		Y			
loop	循环执行某个命令	Y				
losetup	为磁盘设备赋值		Y			
ls	列出符号表			Y		
lwdhcp	自动获取网络地址			Y		
m(m1,m2,m4,m8)	内存显示/修改		Y			
memcpy	从源地址执行拷贝内容到目的地址,支持 64 位地址		Y			
more	分页			Y		
mtest/mt	内存测试					Y
mycmp	比较内存		Y			
mscan	内存漏洞扫描					Y
netboot	网络启动加载		Y			
pcicfg	配置 PCI 空间			Y		
pciscan	列出总线上所有挂在的设备		Y			
pcs	配置 PCI 的功能		Y			
ping	网络建立测试					Y
poweroff	关机				Y	
r	寄存器显示/修改		Y			

reboot	重启 PMON				Y	
search	搜索		Y			
scsiboot	SCSI 磁盘启动加载		Y			
set	设置变量			Y		
sh	命令 Shell			Y		
sleep	PMON 休眠几毫秒				Y	
sty	显示/设置终端洗选项			Y		
sym	设置符号名			Y		
t	跟踪 (single step)		Y			
tftp	启动 tftp 服务			Y		
to	跟踪 (step over)		Y			
tr	透明模式				Y	
unset	删除变量			Y		
version	显示 PMON 版本			Y		
xmodem	Xmodem 协议			Y		
ymodem	Ymodem 协议			Y		

C. 2 命令简表见表 16

表 16 PMON 命令简表

命令	功能及选项	描述
b [-s str] [adr]	显示/设置断点	如果没带参数会打印断点列表，最多支持 32 个软件断点
	[-s str]	当到达断点时，执行字符串的命令
	[adr]	断点的地址
boot [-bsrk] [-o offs][[host:]file]	启动	加载二进制执行文件
	[-b]	禁止清除断点
	[-k]	内核符号
	[-s]	不清除旧的符号表
	[-r]	加载原始文件（raw file）
	[-e]	入口地址，例如板卡 FLASH ROM 的基地址
	[-o]	告诉 FLASH 加载器给镜像分配这个转换地址
	[[host:]file]	网络主机名和文件名
bt [-v] [cnt]	调用栈	显示函数调用栈.
	[-v]	显示函数栈帧（stackframe）基地址和大小
	[cnt]	要显示行的数目.
c [bptadr]	继续执行	从当前停止运行的断点之后继续执行，更新影子寄存器
	[bptadr]	一个的临时断点
call adr [val|-s str]...	调用一个子程序	类似 c 命令继续执行，但不更新影子寄存器
	adr	将被执行的子程序开始地址
	[val]	被传递给函数的值
	[-s str]	传给函数的字符串
compare from to with	比较内存	比较两块内存的内容
	From	用于比较的起始地址
	to	用于比较的结束地址
	with	用于比较的另一个地址的开始地址
copy from to siz	拷贝内存	当拷贝到目标地址低于源地址时，按升序，反之按降序

命令	功能及选项	描述
	from	源的开始地址
	to	目标的开始地址
	siz	要拷贝的内存块大小
d [-b|h|w|s] adr [cnt|-rreg]	显示	datasz 设置默认字大小，moresz 设置默认屏幕
		length
	[-b]	显示 bytes
	[-h]	显示 16-bit words
	[-w]	显示 32-bit words
	[-d]	显示 64-bit words
	[-s]	按空终止字符显示
	adr	显示字符串的基地址
	[cnt]	指定显示的行数
	[-rreg]	显示内存内容当作寄存器 reg 显示
date	显示/设置日期时间 [yymmddhhmm.ss]	显示/设置日期时间
db [numb|*] ...	删除断点	没带参数会列出所有的现存的断点
	[numb]	要删除断点的号码
	[*]	删除所有断点
debug [-svV] [-- args]	进入远程调试模式	
	[-s]	不设置客户端堆栈指针
	[-v]	显示通讯错误
	[-V]	设置详细选项
	[-- args]	设置要传递给客户端程序的参数
devls	列出当前所有设备	列出当前所有设备
dump adr siz [port]	Dump 内存到主机	Dump 内存到主机
	adr	要上传数据的基地址
	siz	要上传的字节数
	[port]	传给这个设备或文件
env	显示环境变量	显示环境变量
eset name	编辑变量	显示变量名并可以编辑（参看 shell）
	[name]	要选择的变量名
fdisk	显示磁盘的分区信息	显示磁盘分区信息
fill from to {val|-s str}...	填充内存	将十六进制模式或字符串写到内存块中，注意 from 必须低于 to 地址
	from	填充操作的基地址
	to	填充操作等结束地址
	[val]	要写到填充区的以 16 进制表示的字节数
	[-s str]	表示内存块应该以 ASCII 字符串填充，包括双引号括起来的多字
flash [-qev] [[[addr] size data]	FLASH EEPROM 操作	编程，擦除，拷贝 FLASH 内存区域，不带参数会给出可用的 FLASH 区域和类型列表
	-q	给出可用的 FLASH 区域和类型列表
	-e base_addr	擦除基地址的 FLASH 区域
	-v addr	检查基地址的 FLASH 区域
	base_addr size from_addr	从 from_addr 开始的内存内容写到 FLASH 的 base_addr 内存地址共写 size 个字节
flush [-di]	刷 Cache	默认同时刷新数据和指令 Cache.

命令	功能及选项	描述
	[-d]	只刷新数据 Cache
	[-i]	只刷新指令 Cache
g [-s] [-b bptadr] [-e adr] [-- args]	Go(开始执行)	默认从 EPC 地址开始并设置栈指针到开始
	[-e adr]	从地址 adr 开始
	[-b bptadr]	设置程序执行的临时断点地址 bptadr
	[-s]	不要设置客户端栈指针
	[-- args]	传递给客户端程序执行的参数
h [*|cmd...]	帮助	默认列出所有的命令
	[*]	提供所有命令的详情
	[cmd]	给出这个命令的帮助
hi [cnt]	显示历史	显示最后的 200 个命令
	[cnt]	显示最近的 cnt 个命令
ifaddr interface ipaddr	配置网卡	配置 网卡的 ip 地址
	interface	网络设备名
	ipaddr	Ip 地址
ifconfig fxp0 [up|down|remove|stat|setmac|readrom|writerom|addr] [netmask]	配置网络	网络配置的各种状态
	fxp0	配置网络 ip 地址，fxp0 为网卡设备号，可以使用 devls 命令查看具体网卡名称
	up	启动指定的网络设备
	down	关闭指定的网络设备
	remove	删除网络设备 IPv6 的 IP 地址
	stat	网络设备状态
	setmac	设置网络设备 mac 地址
	readrom	显示网卡设备上的信息
	writerom	设置网卡设备上的信息
	addr	设置网络设备 IPv6 的 IP 地址
	netmask str	设置网络设备的子网掩码为 str
initrd path	加载镜像	加载 RamDisk 镜像文件
l [-bct] [adr [cnt]]	显示 (反汇编)	默认从 EPC 地址显示反汇编并传给 more
	[-b]	仅列出分支 branch
	[-c]	仅列出调用 calls
	[-t]	列出 trace buffer
	adr	反汇编指令的开始基地址
	cnt	要反汇编的行数
load [-abeist] [-c cmdstr] [-o offset] [-u baud] [port]	从主机下载内存	默认使用当前的波特率
	-y	仅加载符号信息
	-a	禁止加偏移到符号
	-b	禁止在下载前删除所有断点
	-e	禁止清除例外处理程序
	-i	忽略校验错误
	-s	在下载前禁止清除符号表
	-t	在内存的顶部加载
	-f flash_addr -o load_addr offset	往 FLASH 里加载
	-o offset	在指定偏移地方加载
	-r	加载原始文件 (raw file)
	-k	kernel 符号表
	-v	显示详细信息
loop count cmd...	循环执行	命令循环执行某个命令
losetup loopdev0 devfile [bs] [count] [seek]	为 disk 设备赋值	为 disk 设备赋值

命令	功能及选项	描述
ls [-ln] [sym|-[va] adr]	显示符号列表	默认按地址降序列出符号表
	[-l]	提供了一个长的清单，显示每个符号的地址或偏移值
	[-n]	按数字序列表
	[sym]	显示匹配模板的符号。通配符（“？”）词通配符（*）是允许的
	[-v]	是详细信息的选项，显示十六进制、八进制、十进制的值
	[-a]	显示符号表里的地址
	[adr]	显示偏移的地址 adr 的符号
m [adr [hexval|-s str]...]	修改内存	默认进入交互模式，先是当前的地址和值
	adr	是没有使用交互模式时，显示或修改的内存地址
	hexval	设置当前地址值为 hexval 并往前移动一字节
	-s str	拷贝字符串到指定地址
	<CR>	没有参数，进入交互模式，往前移一个字节
	=	在交互模式，重读当前地址
	^|-	在交互模式，后退一个字
	.	退出交互模式
m8 addr [data]	写 double word 的值	在指定地址写入 double word 的值
memcpy src dst count	内存拷贝	内存拷贝
	src	要拷贝内容的源地址，地址可以是 64 位
	dst	要拷贝到的目标地址，地址可以是 64 位
	count	要拷贝内容的大小
more	分屏	（嵌入式命令） 滚动 MORESZ 行. 设置 MORESZ 为 0 取消自动滚动暂停
	<SPACE>	打印一页
	/str	先前搜索字符串 str
	n	重复上一个搜索项
	<CR>	显示下一行 e
	^s|^q	暂停滚动
	q|^c	退出
mtest(mt) [-c] [[addr] size]	内存测试	默认测试所有的内存
	[-c]	执行连续的内存测试
	[addr]	是执行内存测试的基地址
	[size]	执行内存测试的大小
mscan [-s step][-gGl] [start [end]]	内存漏洞扫描	内存漏洞扫描测试程序
	-g	猜测执行 load/store 漏洞探测
	-G	测试执行取指探测
	-l	直接 load/store 探测
	-s step	设置扫描的步长，step 为步长的字节数
mycmp s1 s2 len	比较内存	比较内存和中指定大小的长度
	s1	要比较的内存 s1 起始地址
	s2	要比较的内存 s2 起始地址
	len	要比较的长度
ping [-nqv] [-l preload] [-s size] host	测试网络链接	板卡间的网络包测试

命令	功能及选项	描述
	[-l preload]	在进入正常模式前，首先发送 preload 包
	[-n]	只以数字形式显示地址
	[-q]	在启动和关闭时，只显示摘要行信息
	[-s size]	测试包的大小（默认=56）
	[-v]	除显示 ECHO_RESPONSE 的 “Echo Replies”包外，还列出接收的 ICMP 包
	host	网络主机地址.
pcicfg bus device[:func] reg	PCI 配置	设置/显示 PCI 总线上的设备的寄存器的内容
	bus	总线号
	device	设备号
	:func	功能号
	reg	寄存器
pciscan [-b <bus>][-d <dev>]	PCI 设备列表	列出总线上所有挂在的设备
	[b bus]	-b 选项后跟要显示的 PCI 的总线号
	[d dev]	-d 选项后跟要显示的 dev 号
poweroff	关机	关机
r [reg|* [val|field val]]	显示/设置寄存器值	默认列出通用寄存器
	*	显示出浮点寄存器外的所有寄存器值
	f*	显示所有浮点寄存器值
	reg val	修改寄存器的值为 val
	reg field val	指定寄存器特定域的值
reboot	复位	调到 MIPS 的起始地址 0xBFC00000，和重启 PMON 类似，但他不是完全的硬复位
search from to {val|-s str}.	搜索内存	搜索含在双引号里多字节的字符串
	from	search 操作的起始地址.
	to	search 操作的结束地址.
	val	搜索目标的 16 进制值
..	-s str	搜索操作是个字符串
set [name [value]]	设置和显示环境变量	默认显示所有的环境变量，输入变量名会显示该变量值
	[name]	要设置的环境变量的名字
	[value]	环境变量要设置的字符串
sh	命令 shell	（嵌入式命令） 使用下述的 ASCII 字符输入和特殊的字符执行命令，INBASE 变量设置 shell 默认的数字进制基， 设置 INALPHA 为“hex” 使得把输入作为 16 进制， PROMPT 变量可设置命令提示符。输入字符“!” 会重复先前的命令，设置 RPTCMD 为“on”， 当进入空行时先前的命令会重复，当 RPTCMD 设置为“trace”，只有 trace 命令被重复
	^C	终止命令
	^S	暂停输出流
	^Q	暂停后重新开始输出
	^P	重调先前的命令
	^N	重调下一个命令

命令	功能及选项	描述
	^F	向右移动光标一个字符 (forward)
	^B	向左移动光标一个字符 (back)
	^A	移动光标到行的开始处
	^E	移动光标到行尾
	^D	删除在光标位置的字符
	^H	删除光标左侧的字符
	^K	删除到光标右侧的整行
	;	; 后的输入看作是新命令
	!str	重新调用并执行最后的 str 命令
sh	!num	重调合执行命令号为 num 的命令
	!!	重调并执行最后的命令
	+-/()	代数算子
	^addr	用地址 addr 的内容替代
	@name	用名为 name 寄存器的内容替代
	&name	用符号 name 的值替代
	0xnum	把 num 当 16 进制数
	0onum	把 num 当 8 进制数
	0tnum	把 num 当 10 进制数
sleep ms	PMON 睡眠 n 毫秒	PMON 睡眠 n 毫秒
stty [device] [-va] [baud] [sane] [ter m] [ixany|-ixany] [ixoff|-ixoff]	显示和设置终端选项	默认显示终端类型和波特率
	[device]	是 tty0 或 tty1. 默认是 tty0.
	[-a]	给一个长的列表，显示当前所有的设置
	[-v]	显示波特率和终端类型可能的选择
	[baud]	设置波特率.
	[sane]	复位终端设置为默认值
	[ter m]	设置终端枚举类型
	[ixany]	允许任何字符重启输出
	[-ixany]	只允许 START 重启输出.
	[ixoff]	开启串联模式（tandem mode）
	[-ixoff]	关闭串联模式（tandem mode）
sym name value	定义变量的符号名	定义符号名，可用 ls 显示
	name	要设置值的变量名
	value	要设置变量的值
t [-vbci] [-m adr val] [-M adr val] [-r reg val] [-R reg val] [cnt]	跟踪 (单步)	默认从 EPC 开始
	[-v]	显示每步详情
	[-b]	仅捕获分支
	[-c]	仅捕获函数调用
	[-i]	停止无效的程序计数
	[-m adr val]	当内存地址 adr 和值 val. 相等时停止
	[-M adr val]	当内存地址 adr 和值 val. 不相等时停止
	[-r reg val]	当寄存器 reg 和值 val. 相等时停止
	[-R reg val]	当寄存器 reg 和值 val. 不相等时停止
	[cnt]	跟踪 cnt 个指令然后停止
to [-vbci] [-m adr val] [-M adr val] [-r reg val] [-R reg val] [cnt]	跟踪(step over)	参见 t 命令，只是把函数当作一步
tr [-2] [port]	透明模式	拷贝键盘输入的任何字符到选定的端口，然后拷贝所选端口上来的字符到屏幕

命令	功能及选项	描述
unset name ...	删除变量	使得所有匹配的变量消失
	name	要删除的变量名.支持 *和 ？
vers [-a]	版本号	显示版本号
xmodem [base=baseaddr] [file=filename]	Xmodem 协议	Xmodem 协议支持
	base	目标地址名称
	file	要传送的文件名
ymodem [base=baseaddr] [file=filename]	Ymodem 协议	Ymodem 协议支持

附录 D
（规范性附录）
窗口配置实现

本附录以 LS3A2H 系统为例来介绍窗口配置实现。相关的代码分布在 loongson3_fixup.S、loongson3_HT_init_2h.S、pci_machdep.c、ddr2_config.S。

在此将 CPU 主动发出的访问称为 Downstream 访问，设备主动发出的访问称为 Upstream 访问，则窗口配置的流程如下。

D.1 Downstream 访问配置

由于 PMON 是 32 位的，龙芯上的许多资源都分部在 4GB 以上的物理地址地址空间，因此若要在 PMON 下访问这些资源，只有两种方式，采用 TLB 做地址转换或者通过配置交叉开关直接用 KSEG0/KSEG1 进行访问，显然后者会简单方便。

对于 LS3A 来说一级交叉开关上的每个主设备有 8 个配置窗口，目前 LS3A2H 对此 8 个配置窗口的使用情况是：

a) 6、7、8 三个窗口用于阻挡猜测执行的地址功能；
b) 1 号窗口用于发起对 LS2H CSR Downstream 的访问；
c) 2 号窗口用于发起对 LS2H PCIE 配置寄相关寄存器及 PCIE IO 空间的访问；
d) 3 号窗口用于发起对 LS3A PCI MEM Lo 空间的访问；
e) 4 号窗口用于发起对 LS2H MEM 空间的访问，在 LS3A2H 系统中此空间作为显存用；
f) 5 号窗口用于发起对 LS2H PCIE MEM Lo 空间的访问。

在 PMON 下可用 d8 命令查看 XBAR1 窗口配置情况如下：

```
PMON> d8 0x900000003ff02000 30：
900000003ff02000: 000000001b000000 0000000018000000 ................
900000003ff02010: 0000000010000000 0000000040000000 ...........@....
900000003ff02020: 0000000010000000 00000c0000000000 ................
900000003ff02030: 0000200000000000 0000100000000000 ..... ..........
900000003ff02040: fffffffff000000 fffffffff000000 ................
900000003ff02050: fffffffff000000 fffffffc0000000 ................
900000003ff02060: ffffffff8000000 00000c0000000000 ................
900000003ff02070: 0000200000000000 0000300000000000 ..... .......0..
900000003ff02080: 00000e001f0000f7 00000e00180000f7 ................
900000003ff02090: 0000000010000080 00000e00000000f7 ................
900000003ff020a0: 00000e00100000f7 00000c00000000f7 ................
900000003ff020b0: 00002000000000f7 00001000000000f7 ..... ..........
900000003ff020c0: 0000000000000000 0000000000000000 ................
900000003ff020d0: 0000000000000000 0000000000000000 ................
900000003ff020e0: 0000000000000000 0000000000000000 ................
```

龙芯 3A 的二级交叉开关主要完成地址路由及双通道内存的 interleve 的功能，具体使用情况为：

a) 1 号窗口具有最高的优先级，用于将复位异常地址路由到启动 ROM 即 LPC FLASH;

b) 2 号窗口，用于将 256MB 到 512MB 的物理地址空间路由到除 1 号窗口以外的 IO 空间;

c) 3，4 号窗口，用于 interleve 低端内存;

d) 5，6，7，8 号窗口，用于 interleve 高端内存。

在 PMON 下可用 d8 命令查看 XBAR2 窗口配置情况如下：

```
PMON> d8 0x900000003ff00000 30
900000003ff00000: 000000001fc00000 0000000010000000 ................
900000003ff00010: 0000000000000000 0000000000001000 ................
900000003ff00020: 0000000100000000 0000000100001000 ................
900000003ff00030: 0000000180000000 0000000180001000 ................
900000003ff00040: fffffffffff00000 ffffffff0000000 ................
900000003ff00050: fffffffff0001000 fffffffff0001000 ................
900000003ff00060: ffffffff80001000 ffffffff80001000 ................
900000003ff00070: ffffffff80001000 ffffffff80001000 ................
900000003ff00080: 000000001fc000f2 0000000010000082 ................
900000003ff00090: 00000000000000f0 00000000000000f1 ................
900000003ff000a0: 00000000000000f0 00000000000000f1 ................
900000003ff000b0: 00000000000010f0 00000000000010f1 ................
900000003ff000c0: 0000000000000000 0000000000000000 ................
900000003ff000d0: 0000000000000000 0000000000000000 ................
900000003ff000e0: 0000000000000000 0000000000000000 ................
```

需要注意的是虽然物理地址 0x0 与物理地址 0x10000 0000 都访问低端内存，但是从二级 Cache 看，它们却是不同的 Cache 通路，因此在内核中要避免由于这两段物理地址的混用而导致的 Cache 一致性问题。

D.2 Upstream 的访问配置

龙芯 2H 内部可以发起 DMA 的主设备包括 GPU、DC、PCIE、MEDIA、GMAC、USB、SATA、HDA、DMA 等，LS2H 的结构框图如图 3 所示。除了 GPU、DC 只能访问 DDR 外，其它主设备的请求可以访问路由下表中的所有空间。

表 17 地址空间分配之 DMA 视角

地址空间	目标	说明
0x0000 0000-0x3FFF FFFF	DDR	不维护 IO 一致性
0x4000 0000-0x7FFF FFFF	DDR	维护 IO 一致性
0x8000 0000-0xBFFF FFFF	HT	跨 HT 总线送到 3A 系统内存

对于 LS2H 的每个主设备，在发起 DMA 访问前，都需要分别对自己相应的交叉开关地址路由窗口进行配置，目的是使设备发起的总线地址访问可以到达龙芯 3A 的内存（或其它从设备）。又由于 LS2H 的每个主设备都可以发起 32 位的 DMA 访问(可以全覆盖 4GB 内存空间)，所以可以不使用 SoftIOTLB 功能，以减少一次数据拷贝。为了方便配置，将龙芯 3A 的物理内存映射到 0x10000 0000（4GB）以上的物理地址，这样在设备发出的总线地址前由 HT 窗口或 0x10000 0000，如此则可以直达内存了。

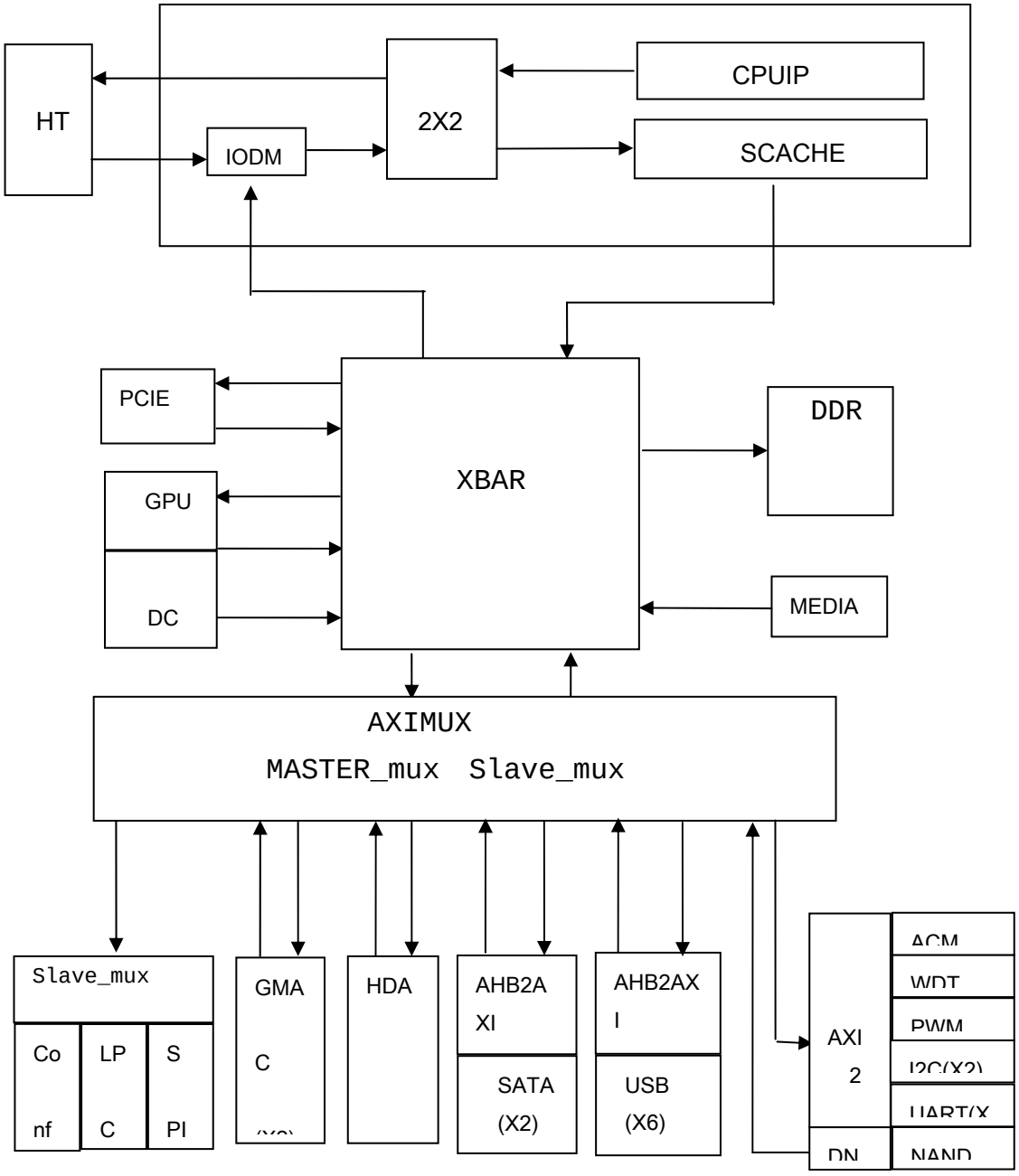


图 3 LS2H 内部的结构框图

表 18 及表 19 分别描述了 LS2H 内部二级交叉开关及一级交叉开关上的主从设备列表及编号：

表 18 二级交叉开关

序号	主设备描述	从设备描述
0	Scache(GS464)	DDR
1	SB	SB
2	GPU	GPU/DC
3	DC	IODMA
4	PCIE	PCIE
5	MEDIA(GS232)	

表 19 一级交叉开关

序号	主设备描述	从设备描述
0	CPUIP(GS464)	Scache
1	HT	IODMA(HT)

D.3 龙芯 3A PCI/PCIX 控制器的窗口配置

除了使用龙芯 3A HT 总线连接南桥 LS2H 外，如果同时使用龙芯 3A 的 PCI 总线扩展设备，此时应注意：CPU 核可以从四个窗口发起 PCI 的 Memory 访问。其中，命中 PCI_MEM Lo0/1/2 的请求在送到 PCI 总线时，地址的高 6 位由相应的 trans_lo 替换；命中 PCI_MEM_Hi 窗口的请求在送到 PCI 总线时，将不作转换。来自 PCI 总线的访问在送到 DDR2 控制器过程中由交叉开关进行地址变换。所以此时还需完成如下操作：

a) 龙芯 3A PCI/PCIX 控制器的初始化，如总线仲裁机制等；
b) 配置一级交叉开关使得：PCI_MEM Lo0 开始的 16M 地址可以到达二级交叉开关；
c) 配置 trans_lo 使得发出去的 PCI MEM 访问到达指定的总线资源区间；
d) 配置 PCIX 的 BAR 和 PCI_Hit_sel 以接收总线上其它主设备发出的请求；
e) 配置二级交叉开关 master1 的窗口路由，使得接收到的设备发出的访问可以到达内存；
f) 驱动程序中直接从 BAR 获得基址访问设备 CSR，同时设备做 DMA 时的地址需要与 4 中的配置一致；
g) PCI 总线上的设备驱动由软件维护 Cache 一致性。

此过程 C 代码实现的例子如下：

```
Targets/Bonito3a2h/pci/pci_machdep.c
//downstream
BONITO_PCIMAP =
     BONITO_PCIMAP_WIN(0, PCI_MEM0_BASE+0x00000000) |
     BONITO_PCIMAP_WIN(1, PCI_MEM0_BASE+0x04000000) |
     BONITO_PCIMAP_WIN(2, PCI_MEM0_BASE+0x08000000) |
     BONITO_PCIMAP_PCIMAP_2;
//upstream receive Bar
#ifdef PCI_DEV_SUPPORT_64BIT//如果 PCI 总线上的设备支持双周期访问，且 PCI 设备驱动也支持 64 位（目前大部分 PCI 设备驱动仅支持 32 位）
   BONITO_PCIBASE0 = 0x00000000;
   BONITO_PCIBASE1 = 0x1;
   BONITO(BONITO_REGBASE + 0x50) = 0xc0000004;
   BONITO(BONITO_REGBASE + 0x54) = 0xeffffff;
//router device access to ddr
   /*set master1's window0 to map pci 4G->DDR 0 */
   asm(".set  mips3;dli  $2,0x900000003ff00100;li  $3,0x100000000;sd
$3,0x0($2);sd        $0,0x80($2);dli        $3,0xffffffff00000000;sd
$3,0x40($2);.set mips0" :::"$2","$3");
```

```
  #else
  BONITO_PCIBASE0 = 0x80000000;
    BONITO_PCIBASE1 = 0xffffffff;
    BONITO(BONITO_REGBASE + 0x50) = 0xc0000000;
    BONITO(BONITO_REGBASE + 0x54) = 0xefffffff;
  /*set master1's window0 to map pci 2G->DDR 0 */
    asm(".set   mips3;dli   $2,0x900000003ff00100;li   $3,0x80000000;sd
$3,0x0($2);sd         $0,0x80($2);dli         $3,0xffffffffc0000000;sd
$3,0x40($2);.set mips0" :::"$2","$3");
  #endif
```

附录 E

（规范性附录）

各板卡 PMON 启动过程

E.1 LS2H 系统启动过程：

1. 设置堆栈；
2. 初始化处理器相关寄存器；
3. 初始化异常向量表；
4. 初始化串口；
5. 设置处理器窗口；
6. 初始化 I2C 控制器；
7. 处理器 Cache 初始化；
8. 配置 DDR；
9. 配置 DDR 16bit；
10. 将代码数据从 FLASH 搬运到内存；
11. 跳转到内存的 C 入口；
12. 解压缩 PMON，跳到 PMON 的真正入口；
13. 利用 RTC 探测处理器频率；
14. 执行构造函数注册数据结构，命令等；
15. 初始化环境变量；
16. 初始化 DC 控制器和配置 FrameBuffer；
17. 停止 USB 复位；
18. 根据 PMON 设备树，依次初始化所有设备，包括 USB 等(如：键盘)；
19. 进入 main 函数；
20. 检查是否设置 al，是则直接加载内核启动，否则进入命令行。

E.2 LS3A2H 系统启动过程

包括 3A 和 2H 两个 BIOS 负责启动过程，

2H 端：

1. 设置堆栈；
2. 初始化处理器相关寄存器；
3. 初始化异常向量表；
4. 初始化串口；
5. 设置处理器窗口；
6. 初始化 I2C 控制器；
7. 处理器 Cache 初始化；
8. 配置 DDR；
9. 配置 DDR 16bit；
10. 将代码数据从 FLASH 搬运到内存；
11. 跳转到内存的 C 入口；

12. 解压缩 PMON，跳到 PMON 的真正入口；
13. 开始轮询。

3A 端：

龙芯 3A 处理器从物理地址 0x1fc00000 处取指执行 PMON 代码，为了简化处理器初始化流程，在 PMON 中指定某个核为 BP(Boot Processor)，完成整个处理器的初始化，其它核为 AP(Application Processor)，AP 在完成了与自身相关的初始化流程后陷入死循环，等待 BP 的唤醒操作；上电后 BP 大致完成如下的操作：

1. 设置堆栈；
2. BP 关 WatchDog；
3. BP 初始化串口；
4. BP 检测 HT 链路；
5. BP 配置窗口寄存器；
6. BP 配置 HT 链路的频率与宽度；
7. 初始化 Cache；
8. 初始化 TLB；
9. BP 配置处理器核；
10. BP 初始化内存控制器；
11. BP 解压缩 PMON，跳转至 C 代码执行；
12. 利用 RTC 探测处理器频率；
13. 执行构造函数注册数据结构，命令等；
14. 初始化环境变量；
15. 初始化 DC 控制器和配置 FrameBuffer；
16. 根据 PMON 设备树，依次初始化所有设备，包括 USB 等；
17. 进入 main 函数；
18. 检查是否设置 al，是则直接加载内核启动，否则进入命令行。

E.3 LS1B 系统启动过程

a） 龙芯 1B 和 MIPS 基础：

1） 龙芯 1B 启动(复位异常)地址是 0xBFC00000，对应于 SPI FLASH 的 0 地址。处理器支持通过内存映射方式访问 0xBFC00000 开始的 1M 地址，PMON 代码放在这部分。

2） 龙芯 1B 复位的时候 DDR 地址位于处理器的 0 地址开始的 256M。如果 DDR 大于 256M，可以重新配置处理器窗口来映射。目前 PMON 设置 1G-2G 的处理器地址窗口来映射到 DDR 的 0-1G。

3） 龙芯 1B 采用 AXI 总线，设备地址固定在 CPU 256M 地址开始的地方。

4） 龙芯 1B 的其他基本情况参考《龙芯 1B 处理器用户手册》。

5） MIPS 上电时候异常向量位于 FLASH 中，PMON 初始化完成后，将中断向量设置到 DDR。

6） MIPS 的 KSEG0，KSEG1 都指向物理地址的 0 开始的 512M，但 Cache 一致性属性不同。KSEG0 是 Cache 访问，KSEG1 是 Uncache 访问。KSEG0 的 Cache 一致性属性取决于 cp0_status 的低 3 位。

7） PMON 代码不使用 TLB，只是将 TLB 初始化，来防治猜测执行访问到这个区域，引起死机。

b) LS1B 系统启动过程

1. 配置 SPI 速度，加快启动速度；
2. 配置 DDR 和处理器时钟；
3. 禁止 GPIO ；
4. 初始化串口 ；
5. 设置处理器窗口；
6. 配置 DDR；
7. 配置 DDR 16bit；
8. 配置百兆 PHY 模式；
9. 处理器 Cache 初始化；
10. 将代码数据从 FLASH 搬运到内存；
11. 跳转到内存的 C 入口；
12. 解压缩 PMON，跳到 PMON 的真正入口；
13. 利用 RTC 探测处理器频率；
14. 执行构造函数注册数据结构，命令等；
15. 初始化环境变量；
16. 初始化 Cache 大小等；
17. 初始化 DC 控制器和配置 FrameBuffer；
18. 停止 USB 复位；
19. 根据 PMON 设备树，依次初始化所有设备，包括 USB 等；
20. 初始化键盘；
21. 将异常向量设置到内存；
22. 初始化 NAND；
23. 进入 main 函数；
24. 检查是否设置 al，是则直接加载内核启动，否则进入命令行。

E.4 LS3A780E/LS3B780E 系统启动过程

Loongson 3 系列处理器(包含 Loongson 3A 和 Loongson 3B1500)上电后所有的处理器核都从物理地址 0x1fc00000 处取指执行 PMON 代码，为了简化处理器初始化流程，在 PMON 中指定某个核为 BP，完成整个处理器的初始化，其它核为 AP，AP 在完成了与自身相关的初始化流程后陷入死循环，等待 BP 的唤醒操作；上电后 BP 大致完成如下的操作：

1. BP 关 WatchDog；
2. BP 初始化 UART；
3. 初始化核内 Cache；
4. BP 检测 HT 链路；
5. BP 配置窗口寄存器；
6. BP 配置 HT 链路的频率与宽度；
7. 初始化核外 Cache；
8. 初始化 TLB；

9. BP 配置处理器核，内存频率，节点频率(针对 Loongson 3B1500)；
10. BP 初始化内存控制器；
11. BP 拷贝压缩的映像文件到内存；
12. BP 解压缩映像文件，跳转至 C 代码执行；
13. 上述操作大部分仅仅由 BP 完成，其它操作则是 BP 与 AP 都需要执行。

几点说明：

1. 对于 Longson 3A 处理器，核内 Cache 包含 L1 指令 Cache 和 L1 数据 Cache，核外 Cache 即 L2 Cache；对于 Loongson 3B1500，核内还包含 Victim Cache，如果把 Victim Cache 当作 L2 Cache，那么核外 Cache 即为 L3 Cache；
2. 相比单路系统，双路（包含 3A 双路和 3B 双路）在 HT 链路配置时除了配置连接北桥的 HT1 控制器，还需要配置连接两颗处理器的 HT0 控制器；
3. PMON C 部分的代码是一个运行在 KSEG0 空间的 32bit 程序，为了访问 32bit 地址以上的 1GB PCI 地址空间，PMON 利用窗口映射将 32bit 以上的 PCI 地址空间映射到物理地址 0x40000000，再利用 BP 的 TLB 将物理地址的 0x40000000～0x7fffffff 映射到 0xc0000000～0xffffffff；
4. PMON 运行时关中断。

附录 F

（规范性附录）

PMON 的 C 代码编程风格

F. 1 说明

PMON 历史代码不强制要求按此约定修改，新写模块自本规范发布之日按此约定执行。本附录用到如下关键字的含义如下：

原则：必须坚持的指导思想

规则：强制必须遵守的约定

建议：必须加以考虑的约定

F. 2 文件

文件概指 .c 和 .h 文件，.h 文件提供接口，.c 文件提供具体实现。合理的头文件布局可以减少编译时间及降低代码维护难度，正确使用头文件可令代码在可读性、文件大小和性能上大为改观，现引入以下方法来帮助合理规划头文件。

F.2.1 规则——头文件 #define 保护

说明：所有头文件都应该使用 #define 防止头文件被多重包含。

示例：loongson3_dev.h

```
#ifndef LOONGSON3_DEF_H_
#define LOONGSON3_DEF_H_
...
#endif
```

F.2.2 规则——头文件中适合放置接口声明，不适合放置实现

说明：头文件是模块(Module)或单元(Unit)的对外接口。头文件中应放置对外部的声明，如对外提供的函数声明、宏定义、类型定义等。

内部使用的函数(相当于类的私有方法)声明不应放在头文件中。

内部使用的宏、枚举、结构定义不应放入头文件中。

变量定义不应放在头文件中，应放在 .c 文件中。

变量的声明尽量不要放在头文件中，亦即尽量不要使用全局变量作为接口。变量是模块或单元的内部实现细节，不应通过在头文件中声明的方式直接暴露给外部，应通过函数接口的方式进行对外暴露。即使必须使用全局变量，也只应当在 .c 中定义全局变量，在 .h 中仅声明变量为全局。

F.2.3 规则——禁止 #include “*.c”

说明：该方法会会使得代码可读性变差以及导致一系列编译问题，如 a.c 中使用了 #include “b.c”，则当 b.c 有更新时，由于 Makefile 中没有显示指定这种依赖，会导致 b.c 的更新没有被编译，增加了调试困难。

F.2.4 建议——每一个 .c 文件应有一个同名 .h 文件，用于声明需要对外公开的接口

说明：如果一个 .c 文件不需要对外公布任何接口，则其就不应当存在，除非它是程序的入口，如 main 函数所在的文件。

其它 .c 文件只能通过包含头文件的方式使用该 .c 提供的接口，禁止在其它 .c 中通过 extern 的方式使用该 .c 提供的外部函数接口、变量，当外部函数接口发生改变时，extern 方式容易导致声明与定义不一致等弊端。

F.2.5 建议——C 文件描述

说明：建议 C 文件按如下示例进行布局，将相同内容相邻放置。

示例：

```
 /************************************************************
对源程序的简要说明，如主要完成功能，独立或依赖关系等
亦可附加作者、完成日期、版本等信息
************************************************************/

#include <stdio.h> //包含所需头文件
...

unsigned int g_device_counts = 0;      //定义全局变量
...

#define MAX_COUNT  128    //定义所需宏
...

static int index = 0; //定义所需本地变量
...

struct struct_name {  //定义本地数据结构
...
};
...

void func (unsigned char, int);    //声明本地函数
...

void func (unsigned char flag, int mask)  //函数实现
{
...
}
...
```

F.3 命名

F.3.1 规则——通用命名规则

说明：统一采用 UNIX Like 风格：单词用小写字母，每个单词直接用下划线“_”分割，例如 text_mutex， kernel_text_address。

函数命名， 变量命名， 文件命名应具备描述性，不要过度缩写。类型和变量应该是名词， 函数名可以用 ″命令性″ 动词(如 open， find 等)。

F.3.2 原则——除了常见的通用缩写以外，不使用单词缩写，不使用汉语拼音

说明：较短的单词可通过去掉“元音”形成缩写，较长的单词可取单词的头几个字母形成缩写，一些单词有大家公认的缩写，常用单词的缩写必须统一。

示例：一些常见可以缩写的例子:

argument	可缩写为	arg
buffer	可缩写为	buff
clock	可缩写为	clk
command	可缩写为	cmd
compare	可缩写为	cmp
configuration	可缩写为	cfg
device	可缩写为	dev
error	可缩写为	err
hexadecimal	可缩写为	hex
increment	可缩写为	inc
initialize	可缩写为	init
maximum	可缩写为	max
message	可缩写为	msg
minimum	可缩写为	min
parameter	可缩写为	para
previous	可缩写为	prev
register	可缩写为	reg
semaphore	可缩写为	sem
statistic	可缩写为	stat
synchronize	可缩写为	sync
temp	可缩写为	tmp

F.3.3 建议——用正确的反义词组命名具有互斥意义的变量或相反动作的函数等

示例:

```
add/remove            begin/end        create/destroy
insert/delete         first/last       get/release
increment/decrement    put/get         add/delete
lock/unlock           open/close       min/max
```

```
old/new                 start/stop      next/previous
source/target           show/hide       send/receive
source/destination      copy/paste      up/down
```

F.3.4 建议——宏命名

说明：采用全大写字母，单词之间加下划线 ”_” 的方式命名(枚举同样建议使用此方式定义)。

F.4 注释

F.4.1 原则——通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的

说明:清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

F.4.2 规则——注释应放在其代码上方相邻位置或右方，不可放在下面。如放于上方则需与其上面的代码用空行隔开，且与下方代码缩进相同

示例:

```
/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000
#define MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */
```

F.4.3 规则——修改代码时，要维护代码周边的所有注释，以保证注释与代码的一致性，不再有用的注释要删除

F.4.4 建议——残留代码

残留代码请及时清除，不要简单使用 #if 0 将其注释掉。若是一些有用的功能代码块，在后续调试、开发过程还需要用到的，请将其用函数方式封装好，并详细注释。

F.5 排版

F.5.1 规则——程序块要采用缩进风格编写，缩进为 4 个空格，不使用制表位(Tab)

说明：把源程序中的 Tab 字符转换成 4 个空格，一个缩进等级是 4 个空格，变量定义和可执行语句要缩进一个等级，函数的参数过长时，也要缩进。

F.5.2 规则——相对独立的程序块之间、变量说明之后必须加空行

示例:

```
//如下例子不符合规范。
if (!valid_ni(ni))
{
// program code
...
```

```
}
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;

//应如下书写
if (!valid_ni(ni))
{
// program code
...
}

repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

F.5.3 规则——多个短语句不允许写在同一行内，即一行只写一条语句

示例:

```
//不好的排版
int a = 5; int b= 10;

//较好的排版
int a = 5;
int b= 10;
```

F.5.4 规则——在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符(如->)，后不应加空格。

说明：采用这种松散方式编写代码的目的是使代码更加清晰。

在已经非常清晰的语句中没有必要再留空格，如括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在C语言中括号已经是最清晰的标志了。在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

示例:

(1) 逗号、分号只在后面加空格，示例：

```
 int a, b, c;
```

(2) 比较操作符，赋值操作符“=”、 “+=”，算术操作符“+”、“%”，逻辑操作符“&&”、“&”，位域操作符“<<”、“^”等双目操作符的前后加空格，示例：

```
if (current_time >= MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

(3) “!”、“~”、“++”、“—”、“&”(地址操作符)等单目操作符前后不加空格，示例：

```
*p = 'a';          // 内容操作"*"与内容之间
flag = !is_empty; // 非操作"!"与内容之间
p = &mem;       // 地址操作"&" 与内容之间
i++;         // "++","--"与内容之间
```

(4) “->”、“.”前后不加空格，示例：

```
p->id = pid;       // "->"指针前后不加空格
```

(5) if、for、while、switch 等与后面的括号间应加空格，使 if 等关键字更为突出、明显，示例：

```
 if (a >= b && c > d)
```

F.5.5 建议——注释符(包括 ‘/*’ ‘//’ ‘*/’)与注释内容之间要用一个空格进行分隔

说明：这样可以使注释的内容部分更清晰。

现在很多工具都可以批量生成、删除‘//’注释，这样有空格也比较方便统一处理。

F.5.6 建议——源程序中关系较为紧密的代码应尽可能相邻

F.5.7 建议——水平留白

说明：水平留白的使用因地制宜，但永远不要在行尾添加没意义的留白。添加冗余的留白会给其他人编辑时造成额外负担，因此，行尾不要留空格。如果确定一行代码已经修改完毕，请将多余的空格去掉。

版本信息

编号：LS-JL423-7-A

版本/状态	编制部门	参与部门	编制/修订历史时间	备注
1.0	系统研发部	系统研发部、芯片研发部、通用事业部、安全应用事业部、嵌入式应用事业部、质量办	2015-03-01	V1.0