LOONGSON

龙芯 2K1000LA 用户手册

2022年11月

手册版本：

V0.2

龙芯中科技术股份有限公司

北京市海淀区中关村环保科技示范园龙芯产业园 100095
Loongson Industrial Park, Zhongguancun Environmental Protection Park,
Haidian District, Beijing 10095, P.R.China
www.loongson.cn

版权声明

免责声明

本文档仅提供阶段性信息，所含内容可根据产品的实际情况随时更新，恕不另行通知。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

龙芯中科技术股份有限公司

Loongson Technology Corporation Limited

地址：北京市海淀区中关村环保科技示范园龙芯产业园 2 号楼

Building No.2, Loongson Industrial Park,

Zhongguancun Environmental Protection Park, Haidian District, Beijing

电话(Tel)：010-

62546668传真(Fax)：

010-62600826

阅读指南

《龙芯2K1000LA用户手册》主要介绍龙芯 2K1000LA相关软件及源码的使用、编译。帮助用户了解和快速搭建应用环境。所涉及包括：PMON源码编译、U-boot源码编译、Linux内核源码编译、文件系统制作、交叉工具链、EJTAG软件使用。

目录

1. 快速开始

本文是龙芯2K1000LA的用户手册，和该手册一起的包括：PMON源码、Linux内核源码、文件系统、交叉工具链、EJTAG软件、龙芯2K1000LA处理器用户手册。本文主要说明上述软件及源码的使用、编译。

1.1 PMON编译环境搭建

将源码压缩包拷贝到工作目录，使用下述命令对源码进行解压。

（注：如果使用的工作环境是虚拟机，请不要直接在共享文件夹下进行解压）

```
1| tar -zxvf pmon-ls2k1000la.tar.gz
```

将交叉工具链拷贝到常用目录，并解压：

```
1| tar -zxvf loongarch64-linux-gnu-2021-12-10-
vector.tar.gz
```

注：本手册以工具链位于/opt/为例

安装编译依赖：

```
1| sudo apt install aptitude xutils-
dev bison flex acpica-tools
```

进入pmon源码编译pmoncfg工具：

该操作仅在源码包解压后的第一次编译前需要执行。

```
1| cd PMON源码/tools/pmoncfg
2| make pmoncfg
3| sudo cp pmoncfg /usr/bin
```

编译PMON：

```
1| cd zloader.ls2k/
2|
3| make cfg all tgt=rom CROSS_COMPILE=/opt/loongarch64-
linux-gnu-2021-12-10-vector/bin/loongarch64-linux-gnu-
 DEBUG=-g
```

```
4| make dtb CROSS_COMPILE=/opt/loongarch64-linux-gnu-
2021-12-10-vector/bin/loongarch64-linux-gnu-
```

编译选项解释：

make cfg 对pmon进行配置；

all为Makefile里的编译项；

tgt=rom，指定tgt为rom，则会生成gzrom.bin文件；

CROSS_COMPILE=loongarch64-linux-gnu-，指定编译工具前缀名；

DEBUG=-g，设置编译的时候携带调试信息。

make dtb ，编译设备树

编译脚本如下，在PMON源码目录下执行：

```
#!/bin/sh
  cd zloader.ls2k/

  make cfg all tgt=rom CROSS_COMPILE=/opt/loongarch64-linux-gnu-2021-12-10-
vector/bin/loongarch64-linux-gnu- DEBUG=-g
  make dtb CROSS_COMPILE=/opt/loongarch64-linux-gnu-2021-12-10-vector/bin/loongarch64-linux-gnu-
  cp gzrom-dtb.bin ../
  cp gzrom.bin ../
  cp ls2k.dtb ../
```

在PMON源码目录生成bin文件和dtb文件，其中gzrom-dtb.bin带dtb，gzrom.bin不带dtb，ls2k.dtb为单独的dtb文件。

例：编译龙芯派PMON

首先编写脚本内容如下：

```
#!/bin/sh
  cd zloader.ls2k/
  sed -i "2c TARGETEL=ls2k_lspi" ./Makefile.ls2k
  make cfg all tgt=rom CROSS_COMPILE=/opt/loongarch64-linux-gnu-2021-12-10-
vector/bin/loongarch64-linux-gnu- DEBUG=-g
  make dtb CROSS_COMPILE=/opt/loongarch64-linux-gnu-2021-12-10-vector/bin/loongarch64-linux-gnu-
  cp gzrom-dtb.bin ../
  cp gzrom.bin ../
  cp ls2k.dtb ../
  sed -i "2c TARGETEL=ls2k" ./Makefile.ls2k
```

然后执行上面的PMON编译脚本，上面的命令将用Targets/ls2k/conf/ls2k_lspi作为配置文件，Targets/ls2k/conf/ls2k_lspi.dts作为dts文件。

后续若仅对DTS文件进行了修改，则可选择不进行make cfg all...这一句命令，只编译设备树文件，最终也会生成新的dtb和包含新dtb的gzrom-dtb.bin。

1.2 LINUX内核编译环境搭建

安装编译依赖：

```
1| sudo apt install libncurses5-dev libssl-dev
```

指定交叉工具链：

```
1| export PATH=/opt/loongarch64-linux-gnu-2021-12-10-
vector/bin:$PATH
```

采用2K1000的配置文件

```
1| cp arch/loongarch/configs/ls2k1000_defconfig .config
2| make menuconfig ARCH=loongarch
```

编译内核：

```
1| make vmlinuz ARCH=loongarch CROSS_COMPILE=loongarch64-
linux-gnu- -j 4
```

编译完成后，会在当前目录下看到生成的vmlinux文件，与压缩后的内核文件vmlinuz。

编译脚本mymake如下：

```
#!/bin/sh
  export LC_ALL=C LANG=C
  export PATH=/opt/loongarch64-linux-gnu-2021-12-10-vector/bin:$PATH
  make vmlinuz ARCH=loongarch CROSS_COMPILE=loongarch64-linux-gnu- -j 4 "$@"
```

编译时执行：

```
1| ./mymake menuconfig
2| ./mymake vmlinuz
```

1.3 文件系统制作

编译前安装依赖参照PMON编译章节所述进行安装。

1.3.1 buildroot文件系统制作

解压buildroot-la.tar.gz文件

```
1| tar -zxvf buildroot-la.tar.gz
```

设置编译配置文件

```
1| cp config-la.my .config
```

或着

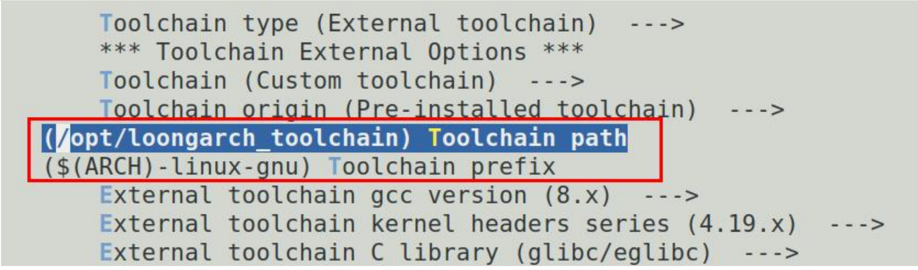
```
1| cp configs/loongisa_defconfig .config
```

指定交叉编译工具链

```
export PATH=/opt/loongarch_toolchain/bin:$PATH
```

打开图形化配置界面，修改工具链路径名，并保存退出

```
make menuconfig ARCH=loongarch64
```



联网编译文件系统

```
make ARCH=loongarch64 CROSS_COMPILE=loongarch64-linux-gnu- -j4 “$@”
```

编译完成后，在buildroot源码的output/images/目录下会生成文件系统镜像文件。

1.3.2 Yocto文件系统制作

解压yocto-la.tar.gz

```
tar zxvf yocto-la.tar.gz
```

解压交叉工具链

```
sudo tar zxvf loongarch-toolchain.tar.gz
```

进入yocto目录，执行下面的命令

```
cd yocto

. oe-init-build-env
```

编译文件系统，执行下面的命令

```
bitbake core-image-minimal
```

1.4 PMON烧录

pmon支持多种烧写方式，具体如下：

1.4.1 调试器更新

linux下直接解压即可使用。

将需要烧录的gzrom-dtb.bin 放入ejtag目录。

进入调试器目录输入如下命令

```
1| sudo ./la_dbg_tool_usb -t
2| source configs/config.ls2k
3| set         #先set后上电，set返回寄存器的值，其中pc需要时
0x1c000000
4| program_cachelock ./gzrom-dtb.bin
```

windows下直接右键解压，然后参考解压出来的文件夹中的doc/ejtag1.pdf安装驱动。

将需要烧录的gzrom-dtb.bin 放入ejtag目录。

直接双击la_dbg_tool_usb启动后使用如下命令烧录。

```
1| source configs/config.ls2k
2| set         #先set后上电，set返回寄存器的值，其中pc需要时
0x1c000000
3| program_cachelock ./gzrom-dtb.bin
```

执行过program_cachelock后打印回到cpu0- 则烧录结束，此时可以重启板卡。

1.4.2 U盘更新

进入PMON，在pmon shell里输入如下命令

```
1| fload (usb0,0)/gzrom-dtb.bin
```

1.4.3 网络更新

需要有tftp服务器，进入PMON，在pmon shell里输入如下命令(假设用2K上的GMAC0口更新)

```
1| ifaddr syn0 ip
2| fload tftp://server-ip/gztom-dtb.bin
```

ip为要设置的该板卡的IP地址，server-ip为连接该板卡的终端机的IP地址

1.4.4 只更新DTB

```
2| load_dtb tftp://server-ip/ls2k.dtb
```

1.5 启动内核

1.5.1手动启动

进入pmon命令行，依次输入如下命令：

```
1| load (wd0,0)/vmlinuz                          #加载内核
2| initrd (wd0,0)/rootfs.cpio.gz                 #加载文件系统
3| g console=ttyS0,115200 rdinit=/sbin/init      #启动
linux
```

此处为参考命令，内核启动支持U盘，硬盘，NAND，网络加载方式。命令路径名称差异详见7章节load.

1.5.2 自动启动

1.5.2.1 使用boot.cfg

PMON启动最后会去常用存储设备的boot目录找boot.cfg文件，并按照boot.cfg 的参数启动，例：

```
timeout 3
default 0
showmenu 1

title 'LoongOS power test'
        kernel (wd0,0)/boot/vmlinuz_test
        args  console=tty console=ttyS0,115200 root=/dev/sda1  mytest=power

title 'LoongOS '
        kernel (wd0,0)/boot/vmlinuz_2kla
        args  console=tty console=ttyS0,115200 root=/dev/sda1 loglevel=8

title 'LoongOS reboot test'
        kernel (wd0,0)/boot/vmlinuz_2kla
        args  console=tty console=ttyS0,115200 root=/dev/sda1 loglevel=8 mytest=reboot
```

其中kernel为内核二进制所在路径，args为内核传参。

注：boot.cfg启动支持U盘，硬盘，网络加载方式。命令路径名称差异详见7章节load.

若U盘与硬盘同时存在boot.cfg系统优先读取使用U盘中的。root用于指定挂载作为文件系统的介质，若仅需使用ramdisk启动则可参考如下boot.cfg

```
timeout 3
default 0
showmenu 1
```

```
title 'LoongOS'
        kernel (wd0,0)/boot/vmlinuz_test
        initrd (wd0,0)/boot/rootfs.cpio.gz
         args  console=tty console=ttyS0,115200
```

1.5.2.2 使用环境变量

如果没有boot.cfg,PMON会执行环境变量autocmd提供的命令，设置autocmd如下：

```
set autocmd "load (wd0,0)/vmlinuz;initrd (wd0,0)/rootfs.cpio.gz;g console=ttyS0,115200 rdinit=/sbin/init"
```

pmon会依次按照autocmd环境变量中的语句来执行，如果没有设置autocmd或者设置的不是启动linux的命令，PMON会加载如下环境变量，并按照环境变量启动内核

```
1| #rd为文件系统路径；al1为内核路径；append为启动参数
2| set rd (wd0,0)/boot/rootfs.cpio.gz
3| set al1 (wd0,0)/boot/vmlinuz
4|set append "g console=ttyS0,115200 rdinit=/sbin/init"
```

（该方案的加载顺序为先执行rd环境变量参数来加载文件系统再使用al1环境变量的参数加载内核，若失败，会使用环境变量al的参数来加载内核）

该方案支持U盘、硬盘、网络方式启动。

1.5.2.3 烧录入NAND后使用环境变量

方案1：

首先使用mkyaffs2工具制作所需的yaffs文件系统

```
1| mkyaffs2 -p 4096 -s 128 --yaffs-ecclayout rootfs/ rootfs.img
```

-p后数字为所使用flash芯片的页大小，-s后数字为flash芯片oobsize。

注：spinand通常oobsize会大一倍，采用硬件ecc时，制作yaffs2文件系统-s指定的为oobsize/2。

依次使用如下命令擦除并烧录

```
1| mtd_erase /dev/mtd0
2| mtd_erase /dev/mtd1
3| devcp tftp://ip/vmlinux /dev/mtd0
4| devcp tftp://ip/rootfs.img /dev/mtd1y
```

> （这里带y的意思就是烧写YAFFS2文件系统）

修改环境变量以自启动

```
6| set al1 /dev/mtd0
7| set append "console=ttyS0,115200 rw
root=/dev/mtdblock1 rootfstype=yaffs2"
```

再次重启即可等待自动加载。

如果是用于引导实时系统的ELF，可以使用方案1对mtd0烧写，mtd1不做操作。

方案2：

首先通过加载内核和ramdisk文件系统的方式进入内核。

在内核下使用如下命令挂载

```
1| Mount /dev/mtdblock0 /mnt/ -t yaffs2
```

并将内核二进制vmlinuz放入其中后

```
2| Umount /mnt/
```

之后使用如下命令挂载

```
1| Mount /dev/mtdblock1 /mnt/ -t yaffs2
```

并将所使用的文件系统放入/mnt/中并展开后

```
2| Umount /mnt/
```

重启回到PMON后使用如下命令修改变量

```
1| set al1 /dev/fs/yaffs2@mtd0/vmlinuz
2| set append "console=ttyS0,115200 rw init=/init
root=/dev/mtdblock1 rootfstype=yaffs2"
```

再次重启即可等待自动加载。

1.6 U-BOOT引导

U-boot相关内容摘自U-boot手册快速开始

1.6.1 u-boot编译

将源码压缩包拷贝到工作目录，使用下述命令对源码进行解压。

（注：如果使用的工作环境是虚拟机，请不要直接在共享文件夹下进行解压）

```
1| tar -zxvf u-boot-la_2022.07.tar.gz
```

将交叉工具链拷贝到常用目录，并解压：

```
1| tar -zxvf loongarch_toolchain.tar.gz
```

注：本手册以工具链位于/opt/为例

指定交叉编译工具链

```
1| export PATH=/opt/loongarch_toolchain/bin:$PATH
```

使用对应板卡的配置文件，此处以LA龙芯派为例

```
1| make loongson2k1000_defconfig
2| cp board/loongson/ls2k1000/config.mk_lspi
board/loongson/ls2k1000/config.mk
2| make menuconfig ARCH=loongarch
```

编译U-BOOT

```
1| make u-boot.bin ARCH=loongarch
     CROSS_COMPILE=loongarch64-linux-gnu- -j4
```

编译完成后，会在当前目录下看到生成的u-boot.bin文件用于烧录。

或直接使用以下脚本进行编译

```
#!/bin/bash
  export PATH=/opt/loongarch_toolchain/bin:$PATH
  case $1 in
        "m")
                make menuconfig ARCH=loongarch
                ;;
        *)
                make u-boot.bin ARCH=loongarch
CROSS_COMPILE=/opt/loongarch_toolchain/bin/loongarch64-linux-gnu- -j4
                cp u-boot.bin ~/tftpboot/u-boot-2k1000la
                ;;
  esac
```

1.6.2 内核编译为uimage

运行的编译脚本compile.sh如下：

请根据使用的交叉工具链以及设备树与文件系统的路径名称进行修改

```
#!/bin/bash
export PATH=/opt/loongarch_toolchain/bin:$PATH
export LC_ALL=C LANG=C
export MAKEFLAGS='CC=loongarch64-linux-gnu-gcc\ -g'
        set -x
        make ARCH=loongarch CROSS_COMPILE=/opt/loongarch_toolchain/bin/loongarch64-
linux-gnu- vmlinux.bin -j4
        lzma -c < arch/loongarch/boot/vmlinux.bin > vmlinux.bin.lzma
        load=$(LC_ALL=C readelf -l vmlinux|grep LOAD |grep -o 0x.*|cut -d' ' -f2)
        load=$(printf "0x%08x 0x%08x" $((($load>>32)&0xffffffff))
$(($load&0xffffffff)))
        entry=$(LC_ALL=C readelf  -e vmlinux|grep Entry|grep -o 0x.*)
        entry=$(printf "0x%08x 0x%08x" $((($entry>>32)&0xffffffff))
$(($entry&0xffffffff)))
        vmlinux=vmlinux.bin.lzma
        rootfs=/home/linux/tftpboot/my_minimal.cpio.gz //文件系统的绝对路径
        dtb=/home/linux/tftpboot/ls2k.dtb           //设备树的绝对路径
```

```
        cat > multi.its << AAA
/*
 * U-Boot uImage source file with multiple kernels, ramdisks and FDT blobs
 */
/dts-v1/;
/ {
    description = "Various kernels, ramdisks and FDT blobs";
    #address-cells = <2>;
    images {
        kernel-1 {
            description = "vmlinux";
            data = /incbin/("$vmlinux");
            type = "kernel";
            arch = "loongarch";
            os = "linux";
            compression = "lzma";
            load = <$load>;
            entry = <$entry>;
        };
        ramdisk-1 {
            description = "rootfs";
            data = /incbin/("$rootfs");
            type = "ramdisk";
            arch = "loongarch";
            os = "linux";
            compression = "none";
        };
        fdt-1 {
            description = "fdt";
            data = /incbin/("$dtb");
            type = "flat_dt";
            arch = "loongarch";
            compression = "none";
        };
    };
    configurations {
        default = "config-1";
        config-1 {
            description = "tqm5200 vanilla-2.6.23 configuration";
            kernel = "kernel-1";
            ramdisk = "ramdisk-1";
            fdt = "fdt-1";
            loadables = "kernel-1", "ramdisk-1";
        };
    };
};
AAA
        mkimage -f multi.its uImage
```

脚本使用方法：

```
./compile.sh
```

编译完成可在运行目录下看到uImage文件。

若文件系统采用硬盘挂载的形式，则需取出上述脚本中关于rootfs的所有描述。

1.6.3 设备树编译

可以使用如下脚本生成DTB文件。

请确保执行该脚本时，当前目录下有ls2k.dts文件和dtc工具。

```
#!/bin/sh
```

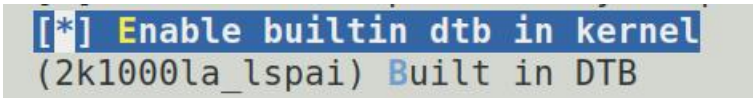
```
 /opt/loongarch64-linux-gnu-2021-12-10-vector/bin/loongarch64-linux-gnu-gcc -mabi=lp64  -
march=loongarch64  -fno-builtin  -E -nostdinc -x assembler-with-cpp -o ls2k.dtb.i ls2k.dts
 ./dtc -I dts -O dtb -o ls2k.dtb  ls2k.dtb.i
```

脚本运行后可在当前目录生成ls2k.dtb

或者使用如下方案放入内核，但需要自行去除上述内核编译脚本中与设备树相关的部分，故此方法并不推荐。

内核中设备树文件位于/arch/loongarch/boot/dts/loongson/

需要打开如下配置




```
1| make dtbs ARCH=loongarch
CROSS_COMPILE=/opt/loongarch_toolchain/bin/loongarch64-
linux-gnu-
```

1.6.4 U-BOOT烧录

1.6.4.1 调试器更新

解压并进入调试器目录输入如下命令，调试器更新请先将需要烧录的文件拷贝到调试器目录

```
1| sudo ./la_dbg_tool_usb -t
2| source configs/config.ls2k
3| set         #先set后上电，set返回寄存器的值
4| program_cachelock ./u-boot.bin
```

1.6.4.2 网络更新

需要有tftp服务器，进入u-boot，在命令行里输入如下命令(假设用2K上的GMAC0口更新)

```
1|setenv serverip 192.168.1.10          #设置主机地址
2|setenv ipaddr 192.168.1.50           #设置板卡地址
3|saveenv                              #保存环境变量设置
```

复位后再次来到u-boot命令行运行如下命令

```
1|tftp 0x9000000008000000 192.168.1.10:u-boot.bin
2|sf probe 0:0 12000 0
3|sf update ${fileaddr} 0x0 ${filesize}
```

注：${fileaddr} 与 ${filesize} 为上一句运行后自动生成，此写法可自动引用不必手动填写具体参数

1.6.5 启动内核

U-BOOT引导启动大致分为两类，

其一：将内核、文件系统与设备树一并制作为一个uImage下载入内存加载，该方案因使用ramdisk，故内存开销较大，但操作方便便于开发验证。

其二：将内核与设备树制作为一个uImage下载入内存加载，文件系统放入硬盘在内核传参中使用root=的方式使内核自动挂载并运行指定硬盘分区中的文件系统，该方案为典型产品启动方案，适合不需要进行文件系统更换的开发或演示。

以下提供部分自动/手动启动方案参考，也可根据用户习惯自行修改操作。

暂不支持通过nfs挂载文件系统。

1.6.5.1自动启动

1.6.5.1.1 网络启动

需要主机端DHCP与TFTP可用。

```
1|setenv serverip 192.168.1.10          #设置主机地址
2|setenv ipaddr 192.168.1.50            #设置板卡地址
3|setenv bootcmd "tftp 0x9000000008000000 uImage;bootm
   0x9000000008000000"
4|setenv bootargs "console=ttyS0,115200
   rdinit=/sbin/init"
5|saveenv                               #保存环境变量设置
```

使用上述命令配置板卡IP并可PING通主机后，默认环境变量是可自动通过网络加载uImage，自动加载时使用的目标文件名称是环境变量bootfile的值。

若使用挂载硬盘文件系统的方式请在bootargs的传参中加入root=/dev/sda1。

（注：这里使用的为龙芯派平台硬盘第一分区，若使用其他平台请确认挂载分区是否正确）

1.6.5.1.2 硬盘启动

```
1|setenv bootcmd "scsi scan;ext4load scsi 0:1
   0x9000000008000000 boot/uImage;bootm
   0x9000000008000000;"
2|saveenv     //之后重启等待自动加载
```

语句中0:1代表scsi scan输出列表中的首个硬盘的第一分区，请确保所使用硬盘的第一分区中有boot/uImage。请注意自己硬盘的文件系统格式若为ext4则命令如上，若使用FAT32则需要使用fatload.

1.6.5.2 手动启动

1.6.5.2.1 网络启动

需要主机端TFTP可用。

```
1|setenv serverip 192.168.1.10          #设置主机地址
2|setenv ipaddr 192.168.1.50            #设置板卡地址
3|setenv bootargs "console=ttyS0,115200
   rdinit=/sbin/init"
4|saveenv                               #保存环境变量设置
```

使用上述命令配置板卡IP并可PING通主机后，加载并启动内核

```
1|tftp 0x9000000008000000 uImage
2|bootm 0x9000000008000000
```

默认环境变量下可输入如下命令启动

```
1|tftp;bootm
```

1.6.5.2.2 硬盘启动

```
1|scsi scan;ext4load scsi 0:1 0x9000000008000000
   boot/uImage;bootm 0x9000000008000000
```

请确保所使用硬盘的第一分区中有boot/uImage。请注意自己硬盘的文件系统格式若为ext4则命令如上，若使用FAT32则需要使用fatload.

2. 接口配置与使用

以下各接口并非所有板卡都有使用，请根据所有板卡的硬件情况参考使用

2.1 GMAC

使用说明

若要同时使用两个网口请确认IP为不同网段

功能测试

请检查所用网口的网口名称，请自行检查所需IP地址

`ifconfig` 可查看当前已唤醒的网络节点

`ifconfig -a`查看所有可用网络节点

注意：PMON下若已经为syn0配置IP且要使用同网段验证另一网口，需要

`ifconfig syn0 remove`后才能对syn1配置IP。

kernel下若已经为eth0配置IP且要使用同网段验证另一网口，需要

`ifconfig eth0 down`后才能对eth1配置IP。

```
PMON> ping 192.168.1.50
PING 192.168.1.50 (192.168.1.50): 56 data bytes
64 bytes from 192.168.1.50: icmp_seq=0 ttl=255 time=0.099 ms
64 bytes from 192.168.1.50: icmp_seq=1 ttl=255 time=0.048 ms

--- 192.168.1.50 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.048/0.060/0.099 ms
PMON> ifconfig syn0 remove
===ioctl sifflags
PMON> ifconfig syn1 192.168.1.50
synopGMAC_linux_open called
Version = 0xd137
MacAddr = 0x30  0x30    0x30    0x30    0x31    0x31

===phy HALFDUPLEX MODE
DMA status reg = 0x0 before cleared!
DMA status reg = 0x0 after cleared!
register poll interrupt: gmac 0
==arp_ifinit done
PMON> No Link: 00000000
Link is up in FULL DUPLEX mode

===phy FULLDUPLEX MODE
Link is with 1000M Speed

PMON> ping 192.168.1.10
PING 192.168.1.10 (192.168.1.10): 56 data bytes
64 bytes from 192.168.1.10: icmp_seq=0 ttl=64 time=0.488 ms
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=0.852 ms
```

```
PMON> ifconfig syn0 192.168.1.50
==arp_ifinit done
PMON> ifconfig syn1 192.168.2.50
synopGMAC_linux_open called
Version = 0xd137
MacAddr = 0x30  0x30    0x30    0x30    0x31    0x31
Link is up in FULL DUPLEX mode

===phy FULLDUPLEX MODE
Link is with 1000M Speed
DMA status reg = 0x0 before cleared!
DMA status reg = 0x0 after cleared!
register poll interrupt: gmac 0
==arp_ifinit done
PMON> ping 192.168.1.10
PING 192.168.1.10 (192.168.1.10): 56 data bytes
64 bytes from 192.168.1.10: icmp_seq=0 ttl=64 time=1.649 ms
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=1.167 ms

--- 192.168.1.10 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 1.167/1.400/1.649 ms
PMON> ping 192.168.2.10
PING 192.168.2.10 (192.168.2.10): 56 data bytes
64 bytes from 192.168.2.10: icmp_seq=0 ttl=64 time=0.637 ms
64 bytes from 192.168.2.10: icmp_seq=1 ttl=64 time=0.854 ms

--- 192.168.2.10 ping statistics ---
```

使用环境变量设置设置MAC地址

配置文件选上`option USE_ENVMAC`，pmon启动后在命令行设置环境变量：

`set ethaddr 00:11:22:33:44:55`

二进制修改生成mac地址：

```
#!/bin/python3
  import struct
  ethaddr = [0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc]
  a = 0x12
  b = 0x15
  c = 0x16
  d = 0x17
  e = 0x18
  with open('gzrom-dtb.bin', "rb") as f:
     f_content = f.read()
     content = struct.unpack("B" * len(f_content), f_content)
  with open('gzrom-dtb_1.bin','wb') as f_w:
     f_w.write(f_content)
     f_w.seek(0xff1ee)
     f_w.write(struct.pack("B" * 6, ethaddr[0],
ethaddr[1],ethaddr[2],ethaddr[3],ethaddr[4],ethaddr[5]))
```

EEPROM存储MAC地址

eeprom存储mac地址， pmon启动自动读取eeprom内存储的mac地址，如果没有存或者存的值不符合要求，pmon会生成随机地址

pmon下设置mac地址的命令：`setmac syn1 "00:11:22:33:44:55"`

2.2 UART

使用说明

12个UART的使用数量与模式见处理器用户手册通用配置寄存器1 位域0-13

232测试连接使用RS232转USB

485测试请使用USB转RS485转换器

检测内核配置项`SERIAL_8250_NR_UARTS`和`SERIAL_8250_RUNTIME_UARTS`设置的数值是否大于等于要使用的数值

功能测试

内核下使用如下语句（x是要操作的串口号，对应关系见说明）

`stty -F /dev/ttySx speed 115200 cs8` 修改串口波特率与数据位

`stty -F /dev/ttySx` 查看串口属性

ttyS1-4进行设置

输出测试：

`echo “123” > /dev/ttyS1` 对应测试串口会输出123的打印

输入测试：

`cat /dev/ttyS1` 对应测试串口输入任意内容后回车可在调试串口看到打印

2.3 SPI

使用说明

不同板卡SPI片选对应的器件不一样，

一般SPI0 片选0 为 nor-flash 用于存放PMON

功能测试

PMON下

`set_dev_pins spi0`选用要操作的SPI控制器(根据版本差异，可能不支持该命令)

`spi_base 0`选用要操作的

`spi_id` 读取设备信息

`read_pmon 0 0x100`从0x0开始读取PMON0x100字节

```
PMON> spi_base 0
PMON> spi_id
Manufacturer's ID:          c8
Device ID-memory_type:      40
Device ID-memory_capacity: 14
PMON> read_pmon 0 0x100

0x00000000    0c fe 3f 14 8c 81 88 03   8c 01 20 03 0d fc 83 03
0x00000010    8d 15 00 29 0d 5c 80 03   8d 11 00 29 0c 02 00 14
0x00000020    8c 01 08 04 0c 00 88 03   8c 01 02 04 0c 3c 80 03
0x00000030    8c 01 20 03 2c 00 06 04   0c 7c 80 03 8c 01 24 03
0x00000040    2c 04 06 04 2c 00 00 14   cc 1f 00 16 8c 01 20 03
0x00000050    0d c4 3f 14 ad 01 20 03   8d 41 80 29 8e 11 80 28
0x00000060    ce 09 80 03 8e 11 80 29   00 70 5f 54 00 00 40 03
0x00000070    0c 0e 00 14 80 11 00 04   2c 00 38 14 8c 01 24 03
0x00000080    2c 30 00 04 2c 00 38 14   2c 20 02 04 0c 20 80 03
0x00000090    80 05 03 04 0c 10 80 03   80 01 00 04 c3 01 00 1c
0x000000a0    63 30 d2 28 c2 01 00 1c   42 d0 d1 28 0d 00 24 03
0x000000b0    00 04 00 54 2c 30 c0 02   ac 31 15 00 80 01 00 4c
0x000000c0    0c c0 82 03 2c 00 00 04   d7 01 00 1c f7 02 d2 28
0x000000d0    04 00 38 14 84 00 20 03   f7 92 11 00 c3 01 00 1c
0x000000e0    63 30 d1 28 c2 01 00 1c   42 d0 d0 28 0c 80 00 04
0x000000f0    8c fd 4f 03 04 00 24 03   8d 0d 40 03 ae 49 41 00
```

内核下

某些板卡SPI0片选1会有用于存贮的nand-flash芯片，内核选用MTD支持。

内核下/dev下会有一个16MB nand-flash模拟出的块设备节点mtdblock0

```
root@ls3a5000:~# ls /dev/
block             fd        memory_bandwidth
bsg               full      mpt2ctl
bus               i2c-0     mpt3ctl
char              i2c-1     mtab
console           initctl   mtd0
core              input     mtd0ro
cpu_dma_latency   kmsg      mtdblock0
```

使用如下语句

```
mount /dev/mtdblock0 /mnt -t jffs2
```

将块设备成功挂载到/mnt后可以进行读写。

```
root@ls3a5000:~# mount /dev/mtdblock0 /mnt/ -t jffs2
[   92.090221] jffs2: notice: (340) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0 of xdatum (0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.
root@ls3a5000:~# ls /mnt/
test.c
```

（若挂载失败，`flash_erase /dev/mtd0 0 0` 擦除一下，再挂载）

其余片选会以spidev0,x的形式出现在/dev下

2.4 OTG

使用说明

OTG接口为micro-B 主从双模式自动检测，该测试项主模式使用micro-B转母A接U盘，从模式将设备模拟为U盘，也可更改内核配置使其模拟为网口或打印机。

功能测试

主模式测试：

内核下micro-B转母A接U盘后内核识别并创建设备节点

`mount /dev/sdb1 /mnt` 挂载U盘后可读写U盘

```
usb 1-1: new high-speed USB device number 2 using dwc2
usb-storage 1-1:1.0: USB Mass Storage device detected
scsi host1: usb-storage 1-1:1.0
[drm] LS2K1000 found
[drm] port1 is defer probed
scsi 1:0:0:0: Direct-Access     Kingston DataTraveler 2.0 PMAP PQ: 0 ANSI: 6
[drm] LS2K1000 found
[drm] port1 is defer probed
sd 1:0:0:0: [sdb] 15249408 512-byte logical blocks: (7.81 GB/7.27 GiB)
sd 1:0:0:0: [sdb] Write Protect is off
sd 1:0:0:0: [sdb] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
 sdb: sdb1
sd 1:0:0:0: [sdb] Attached SCSI removable disk
random: crng init done

~# mount /dev/sdb1 /mnt/
FAT-fs (sdb1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
~#
```

从模式测试：

内核下运行otg.sh脚本 ,micro-B转公A接电脑，可识别到一个32M的U盘可以读写。

```
root@ls3a5000:~# ./otg.sh
mount: mounting none on /sys/kernel/config failed: Device or resource busy
[  384.647344] Mass Storage Function, version: 2009/09/11
[  384.652570] LUN: removable file: (no medium)
32+0 records in
32+0 records out
33554432 bytes (34 MB, 32 MiB) copied, 0.290322 s, 116 MB/s
mkfs.fat 4.2 (2021-01-31)
40000000.otg
[  384.968800] dwc2 40000000.otg: bound driver configfs-gadget
```

otg.sh内容如下

```
#!/bin/sh

#挂载configfs文件系统
mount -t configfs none /sys/kernel/config
cd /sys/kernel/config/usb_gadget
mkdir g1
cd g1

#设置USB协议版本
echo 0x0200  > bcdUSB

#定义产品的VendorID和ProductID
echo "0xABCD" >idVendor
echo "0x1017" >idProduct

#实例化"英语"ID:
mkdir strings/0x409

#将开发商、产品和***字符串写入内核:
echo "012345678ABCDEF" >strings/0x409/serialnumber
echo "Dragon" >strings/0x409/manufacturer
echo "DragonMSC"  >strings/0x409/product

#创建一个USB配置实例:
mkdir configs/config.1

echo 120 > configs/config.1/MaxPower

#定义配置描述符使用的字符串
mkdir configs/config.1/strings/0x409
echo "mass_storage" >   configs/config.1/strings/0x409/configuration

#创建一个功能实例，需要注意的是，一个功能如果有多个实例的话，扩展名必须用数字编号:
mkdir functions/mass_storage.0

dd if=/dev/zero of=/mnt/disk.img bs=1M count=32
mkfs.vfat /mnt/disk.img

#配置U盘参数
echo "/mnt/disk.img" > functions/mass_storage.0/lun.0/file
```

```
echo 1 > functions/mass_storage.0/lun.0/removable
echo 0 > functions/mass_storage.0/lun.0/nofua

#捆绑功能实例到配置config.1
ln -s functions/mass_storage.0 configs/config.1

#查找本机可获得的UDC实例
ls /sys/class/udc/
#40000000.otg

#将gadget驱动注册到UDC上，插上USB线到电脑上，电脑就会枚举USB设备。
echo "40000000.otg" > UDC
```

2.5 IIC

使用说明

请首先确认所用平台各I2C总线下挂设备的数量与地址

功能测试

PMON下

以挂载挂载了EEPROM为例

`eepread 0 5` 从0地址读取EEPROM五字节内存

`Eepread 0 “11 22 33 44 55”` 从0地址按字节写入

```
PMON> eepread 0 5
   0: 0xff
   1: 0xff
   2: 0xff
   3: 0xff
   4: 0xff
PMON> eepwrite 0 "11 22 33 44 55"
   0 <= 0x11
   1 <= 0x22
   2 <= 0x33
   3 <= 0x44
   4 <= 0x55
PMON> eepread 0 5
   0: 0x11
   1: 0x22
   2: 0x33
   3: 0x44
   4: 0x55
PMON>
```

内核下

`i2cdetecte -r -y 0`可以扫描对应总线的设备挂载情况

```
# i2cdetect -r -y 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- UU -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
#
```

`date` 读取外置RTC时间

注意内核配置是使用的内部还是外部时钟，实在不清楚就把DTS中RTC节点删掉，这样肯定用的就是外部时钟。

```
root@ls3a5000:~# date
Thu Sep  8 11:13:39 UTC 2022
```

`echo 123 > /sys/bus/i2c/devices/1-0050/eeprom` 可对写入EEPROM

`cat /sys/bus/i2c/devices/1-0050/eeprom` 可查看EEPROM内容

若要验证容量，可以将其写入文件

`echo `cat /sys/bus/i2c/devices/1-0050/eeprom` > test.c`

然后使用`wc -c test.c` 查看读取的字符数量是否正确

```
root@ls3a5000:~# wc  -c test.c
131073 test.c
```

2.6 SATA

使用说明

建议根据2K1000LA用户手册制作启动盘使用

功能测试

内核下

`monut /dev/sda1 /mnt` 可正常读写

建议根据2K1000LA用户手册制作启动盘使

2.7 CAN

使用说明

使用CAN0 CAN1

功能测试

将CAN0和CAN1的引脚对应相连

运行can_test.sh脚本，若就收放收到信号则正常

```
root@ls3a5000:~# ./can_test.sh
+ /sbin/ip link set can0 type can bitrate 100000
[  236.342112] sja1000_platform 1fe20c00.can can0: setting BTR0=0x18 BTR1=0x7f
+ sleep 1s
+ /sbin/ip link set can1 type can bitrate 100000
[  237.361908] sja1000_platform 1fe20d00.can can1: setting BTR0=0x18 BTR1=0x7f
+ sleep 1s
+ ifconfig can0 up
[  238.379452] IPv6: ADDRCONF(NETDEV_UP): can0: link is not ready
[  238.385372] IPv6: ADDRCONF(NETDEV_CHANGE): can0: link becomes ready
+ sleep 1s
+ ifconfig can1 up
[  239.401430] IPv6: ADDRCONF(NETDEV_UP): can1: link is not ready
[  239.407357] IPv6: ADDRCONF(NETDEV_CHANGE): can1: link becomes ready
+ sleep 1s
+ sleep 1s
+ candump can0
interface = can0, family = 29, type = 3, proto = 1
+ cansend can1 -i 12 0x11 0x22 0x33 0x44 0x55 0x66 0x77
interface = can1, family = 29, type = 3, proto = 1
<0x00c> [7] 11 22 33 44 55 66 77
root@ls3a5000:~#
```

can_test.sh内容如下

```
#!/bin/sh

ifconfig can0 down
sleep 1s
ifconfig can1 down
sleep 1s
set -x
/sbin/ip link set can0 type can bitrate 100000
sleep 1s
/sbin/ip link set can1 type can bitrate 100000
sleep 1s
ifconfig can0 up
sleep 1s
ifconfig can1 up
sleep 1s
candump can0 &
sleep 1s
cansend can1 -i 12 0x11 0x22 0x33 0x44 0x55 0x66 0x77
```

2.8 GPIO

使用说明

gpio部分有引出过灯，可由灯的亮灭确认输出，其余直接引出引脚的GPIO建议用外接灯或万用表的形式确认其是否输出与操作一致

2k1000LA芯片 GPIO0-3 有独立中断号，可使用脉冲触发，GPIO4-31公用一个中断号 GPIO32-64共用一个中断号，需要在处理函数中通过寄存器状态来判读中断的GPIO，只可高电平触发。

功能测试

PMON下可通过mx/dx命令来查看和修改对应寄存器值来控制GPIO例如：

```
m4 0x1fe00500 0x0
```

```
m4 0x1fe00510 0x0
```

此时GPIO全低

【内核下测试方案】

方案1：直接操作寄存器，该方案需要devmem工具

以GPIO1为例，需要结合芯片手册来了解GPIO所对应的位

设置输出

```
# devmem 0x1fe00500
0xFFFFFFF7
# devmem 0x1fe00500 w 0xfffffff5
#
# devmem 0x1fe00510 w 0x0
```

读取输入

```
# devmem 0x1fe00500
0xFFFFFFF5
# devmem 0x1fe00500 w 0xfffffff7
# devmem 0x1fe00520
0x7FF00007
```

方案2：通过GPIO驱动calss操作

以GPIO1为例

```
# echo 1 > /sys/class/gpio/export
# cat /sys/class/gpio/gpio1/value
1
# echo out > /sys/class/gpio/gpio1/direction
# echo 1 > /sys/class/gpio/gpio1/value
# echo 0 > /sys/class/gpio/gpio1/value
```

方案3：驱动测试，该驱动仅用于测试，GPIO的使用建议参考下方应用方案。

只修改gpio_set_info即可，仅GPIO0-3可修改触发方式，测试时请勿将中断引脚悬空，以防因干扰导致中断误触

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/sched.h>
#include <linux/pm.h>
#include <linux/sysctl.h>
#include <linux/proc_fs.h>
```

```
#include <linux/delay.h>
#include <linux/gpio.h>
#include <loongson-2k.h>

#define GPIO_INT_ENABLE 0x1fe00530
#define GPIO_IN_OUT_SET 0x1fe00500
#define GPIO_OUT_VALUE  0x1fe00510
#define INT_POL         0x1fe01470
#define INT_EDGE        0x1fe01474
#define INT_CLR         0x1fe0146c
#define INT_SET         0x1fe01468

#define SET_VALUE   0
#define SET_IRQ     1

#define LOW         0
#define HIGH        1

struct ls2k_gpio_info
{
    int     gpio;
    int     irq;
    int     irq_flags;
    int     type;
    int     value;
    char    name[6];
};

struct ls2k_gpio_info  gpio_set_info[]={
/*  [0]={
        .gpio=3,
        .type=SET_VALUE,
        .value=LOW
    },
    [1]={
        .gpio=2,
        .type=SET_VALUE,
        .value=LOW
    },
    [2]={
        .gpio=21,
        .type=SET_VALUE,
        .value=LOW
    },
    [3]={
        .gpio=1,
        .type=SET_IRQ,
    },
    [4]={
        .gpio=29,
        .type=SET_IRQ,
    },
    [5]={
        .gpio=27,
        .type=SET_IRQ,
```

```
    },
    */
};
int max=ARRAY_SIZE(gpio_set_info);
static int gpio_num ;

static int ls2k_gpio_info(struct ls2k_gpio_info *dev)
{
    char offset;
    struct ls2k_gpio_info *date = dev;

    if(date->gpio < 4) {
        offset = 28 + date->gpio;
    } else if(date->gpio < 32) {
        offset = 26;
    } else {
        offset = 27;
    }

    sprintf(date->name, "gpio%d",date->gpio);

    printk("text %s irq:%d\n",date->name,date->irq);

    if(IRQF_TRIGGER_RISING & date->irq_flags) {
        ls2k_writel(ls2k_readl(INT_EDGE) | (1 << offset),  INT_EDGE);       //set edge as pluse
        ls2k_writel(ls2k_readl(INT_POL)  & ~(1 << offset), INT_POL);        //set pol down
    } else if(IRQF_TRIGGER_FALLING & date->irq_flags) {
        ls2k_writel(ls2k_readl(INT_EDGE) | (1 << offset),  INT_EDGE);       //set edge as pluse
        ls2k_writel(ls2k_readl(INT_POL)  | (1 << offset),  INT_POL);        //set pol up
    } else if(IRQF_TRIGGER_HIGH & date->irq_flags) {
        ls2k_writel(ls2k_readl(INT_EDGE) & ~(1 << offset), INT_EDGE);       //set edge as level
        ls2k_writel(ls2k_readl(INT_POL)  & ~(1 << offset), INT_POL);        //set pol low
    } else if(IRQF_TRIGGER_LOW & date->irq_flags) {
        ls2k_writel(ls2k_readl(INT_EDGE) & ~(1 << offset), INT_EDGE);       //set edge as level
        ls2k_writel(ls2k_readl(INT_POL)  | (1 << offset),  INT_POL);        //set pol high
    } else {
        return -1;
    }
    ls2k_writel(ls2k_readl(GPIO_IN_OUT_SET + date->gpio/32*4) | (1 << date->gpio%32),
GPIO_IN_OUT_SET + date->gpio/32*4);   // INPUT TYPE
    ls2k_writel(ls2k_readl(GPIO_INT_ENABLE + date->gpio/32*4) | (1 << date->gpio%32),
GPIO_INT_ENABLE + date->gpio/32*4);   // irq enable

    return 0;
}

static irqreturn_t gpio_handler(int irq, void *dev)
{
    int value = 0;
    char offset;
    struct ls2k_gpio_info *p = (struct ls2k_gpio_info *)dev;

    if(p->gpio < 4) {
        offset = 28 + p->gpio;
    } else if(p->gpio < 32) {
```

```
        offset = 26;
    } else {
        offset = 27;
    }

    if((p->gpio) >=4) {
        value = gpio_get_value(p->gpio);
        if(p->irq_flags & IRQF_TRIGGER_HIGH) {
            if(!value)
                return IRQ_NONE;
        } else if(p->irq_flags & IRQF_TRIGGER_LOW) {
            if(value)
                return IRQ_NONE;
        } else {
            return IRQ_NONE;
        }
    }

    printk("IRQ-%s\n",p->name);

    return IRQ_HANDLED;
}

static int __init gpio_set_info_init(void)
{
    int error;
    int num=0;

    for(num=0;num<max;num++)
    {
        gpio_num=gpio_set_info[num].gpio;
        if((gpio_num < 0) || (gpio_num > 63))
        {
            printk("gpio_num unknown\n");
            return -1;
        }
        if(gpio_set_info[num].type==SET_VALUE)
        {
            ls2k_writel(ls2k_readl(GPIO_IN_OUT_SET + gpio_num/32*4) &  ~(1 << gpio_num%32),
GPIO_IN_OUT_SET + gpio_num/32*4);   // OUTPUT TYPE
            if(gpio_set_info[num].value==HIGH)
                ls2k_writel(ls2k_readl(GPIO_OUT_VALUE + gpio_num/32*4) | (1 << gpio_num%32),
GPIO_OUT_VALUE + gpio_num/32*4);   //value
            else
                ls2k_writel(ls2k_readl(GPIO_OUT_VALUE + gpio_num/32*4) & ~(1 << gpio_num%32),
GPIO_OUT_VALUE + gpio_num/32*4);
            continue;
        }
        else if(gpio_set_info[num].type==SET_IRQ)
        {

            gpio_set_info[num].irq=gpio_to_irq(gpio_num);
            if(gpio_num < 4) {
                gpio_set_info[num].irq_flags = IRQF_TRIGGER_FALLING;
            } else {
```

```
                gpio_set_info[num].irq_flags = IRQF_SHARED | IRQF_TRIGGER_HIGH;
            }

            error = ls2k_gpio_info(&gpio_set_info[num]);
            if (error < 0) {
                printk("gpio-irq: failed to ls2k_gpio_info"
                        " for GPIO %d, error %d\n",gpio_num, error);
                goto fail1;
            }

            error = request_irq(gpio_set_info[num].irq, gpio_handler,
gpio_set_info[num].irq_flags, gpio_set_info[num].name,&gpio_set_info[num]);
            if (error) {
                printk("gpio-irq: Unable to claim irq %d; error %d\n",gpio_set_info[num].irq,
error);
                goto fail2;
            }
        }
        else{
            printk("gpio %d,unknown type\n",gpio_num);
        }
    }
    return 0;

fail2:
    for(;num>=0;num--)
    {

        if(gpio_set_info[num].type==SET_VALUE)
            continue;
        else
        {
            gpio_num=gpio_set_info[num].gpio;
            free_irq(gpio_set_info[num].irq,&gpio_set_info[num]);
            ls2k_writel(ls2k_readl(GPIO_INT_ENABLE + gpio_num/32*4) & ~(1 << gpio_num%32),
GPIO_INT_ENABLE + gpio_num/32*4);    //irq disable
        }
    }
fail1:
    return error;
}

static void __exit gpio_set_info_exit(void)
{

    int num=0;
    for(num=0;num<max;num++)
    {
        if(gpio_set_info[num].type==SET_VALUE)
            continue;
        else if(gpio_set_info[num].type==SET_IRQ)
        {
            gpio_num=gpio_set_info[num].gpio;
            free_irq(gpio_set_info[num].irq,&gpio_set_info[num]);
```

```
            ls2k_writel(ls2k_readl(GPIO_INT_ENABLE + gpio_num/32*4) & ~(1 << gpio_num%32),
GPIO_INT_ENABLE + gpio_set_info[num].gpio/32*4); //irq disable
        }
        else
        {
            printk("gpio %d,unknown type\n",gpio_num);
        }
    }
}

module_init(gpio_set_info_init);
module_exit(gpio_set_info_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("loongson <xxx@loongson.cn>");
MODULE_DESCRIPTION("ls2k GPIO irq");
MODULE_ALIAS("LS2K");
```

【内核下应用方案】

方案1：gpio-leds驱动的使用

设备树对应结点为

```
    leds {
            compatible = "gpio-leds";
        led1{
            label = "led1";
            gpios = <&pioA 27 0>;
        };
        led2{
            label = "led2";
            gpios = <&pioA 20 0>;
        };
    };
```

内核配置打开Device Drivers->LED Support中

```
<*>   LED Support for GPIO connected LEDs
```

使用`echo 1 > /sys/class/leds/led1/brightness` 进行控制

方案2：gpio_keys_polled驱动的使用

设备树对应结点为

```
  key-polled{
      compatible = "gpio-keys-polled";
      poll-interval = <100>;
      key1{
            gpios = <&pioA 1 0>;
            code = <2>;
            label ="key_gpio1";
        };
      key2{
            gpios = <&pioA 2 0>;
            code = <3>;
            label ="key_gpio2";
        };
    };
```

内核配置打开Device Drivers->Input device support->Keyboards中

```
<*>   GPIO Buttons
<*>   Polled GPIO buttons
```

需要修改内核驱动gpio_keys_polled.c此处为

```
pdata->rep = device_property_present(dev, "autorepeat");
device_property_read_u32(dev, "poll-interval", &pdata->poll_interval);

device_for_each_child_node(dev, child) {
    if (fwnode_property_read_u32(child, "code",
                    &button->code)) {
        dev_err(dev, "button without keycode\n");
        fwnode_handle_put(child);
        return ERR_PTR(-EINVAL);
    }
```

之后就可以在内核下使用`hexdump /dev/input/event0` 并操作按键

```
00000c0 57fc 6333 0000 0000 1904 000d 0000 0000
00000d0 0001 0003 0001 0000 57fc 6333 0000 0000
00000e0 1904 000d 0000 0000 0000 0000 0000 0000
00000f0 5801 6333 0000 0000 a0c0 0006 0000 0000
0000100 0001 0003 0000 0000 5801 6333 0000 0000
0000110 a0c0 0006 0000 0000 0000 0000 0000 0000
```

可以看到键值与按键状态的上报。

2.9 PWM

使用说明

一般PWM控制器与GPIO复用会引出接灯，可通过控制占空比看灯的闪烁。

功能测试

pwm测试脚本

```
#!/bin/sh

#pwm0 square wave output

devmem2 0x1fe22008 w 0x1000000  // Base+0x8 Pulse period buffer register
devmem2 0x1fe22004 w 0x7fffff   // Base+0x4 Low pulse buffer register
devmem2 0x1fe2200c w 0x1        // Base+0xc Control register
```

2.10 HDA

使用说明

请确认所用板卡使用的是否为HDA接口

功能测试

测试以2k1000LA龙芯派为例

进入内核运行hda.sh来配置寄存器，脚本内容如下：

```
#!/bin/bash

amixer sset 'Master' 100% unmute
amixer cset numid=14 , iface=MIXER,name='Input Soure' 1
amixer sset 'Capture', 0 100% unmute
amixer sset 'Capture', cap

amixer sset 'Rear Mic',0 31
amixer sset 'Rear Mic Boost',0 3
#you can use alsamixer to config as well.
```

```
root@ls3a5000:~# ./hda.sh
numid=14,iface=MIXER,name='Input Source'
  ; type=ENUMERATED,access=rw------,values=1,items=3
  ; Item #0 'Front Mic'
  ; Item #1 'Rear Mic'
  ; Item #2 'Line'
  : values=1
Wrong scontrol identifier: Capture,
Simple mixer control 'Capture',0
  Capabilities: cvolume cswitch
  Capture channels: Front Left - Front Right
  Limits: Capture 0 - 63
  Front Left: Capture 0 [0%] [-17.25dB] [on]
  Front Right: Capture 0 [0%] [-17.25dB] [on]
Simple mixer control 'Rear Mic',0
  Capabilities: pvolume pswitch
  Playback channels: Front Left - Front Right
  Limits: Playback 0 - 31
  Mono:
  Front Left: Playback 31 [100%] [12.00dB] [off]
  Front Right: Playback 31 [100%] [12.00dB] [off]
Simple mixer control 'Rear Mic Boost',0
  Capabilities: volume
  Playback channels: Front Left - Front Right
  Capture channels: Front Left - Front Right
  Limits: 0 - 3
  Front Left: 3 [100%] [30.00dB]
  Front Right: 3 [100%] [30.00dB]
root@ls3a5000:~# aplay whatarewords.wav
```

`aplay my.wav` 播放my.wav音频文件

`arecord -d 5 -f cd -t wav test.wav` 录音后可以用于播放

2.11 I2S

使用说明

请确认所用板卡使用的是否为I2S接口

功能测试

测试以2k1000LA工业派为例

无需手动配置

`aplay my.wav` 播放my.wav音频文件

`arecord -d 5 -f cd -t wav test.wav` 录音后可以用于播放

2.12 外置看门狗

使用说明

外置看门狗一般状态是关闭的，使能用GPIO44，喂狗使用GPIO45，与NAND为复用关系。

功能测试

PMON下

`m4 0x1fe00504 0xffffcf6f` 使能

交替运行以下语句来喂狗

`m4 0x1fe00514 0x00003000`

`m4 0x1fe00514 0x00001000`

停止喂狗约6S触发复位

3. PMON板卡适配

（注：PMON v1.0 以上版本可用）

3.1 LA板卡适配

3.1.1 自动适配

3.1.2 手动适配

3.2 MIPS板卡适配

3.2.1 自动适配

3.2.2 手动适配

4. 应用程序编译

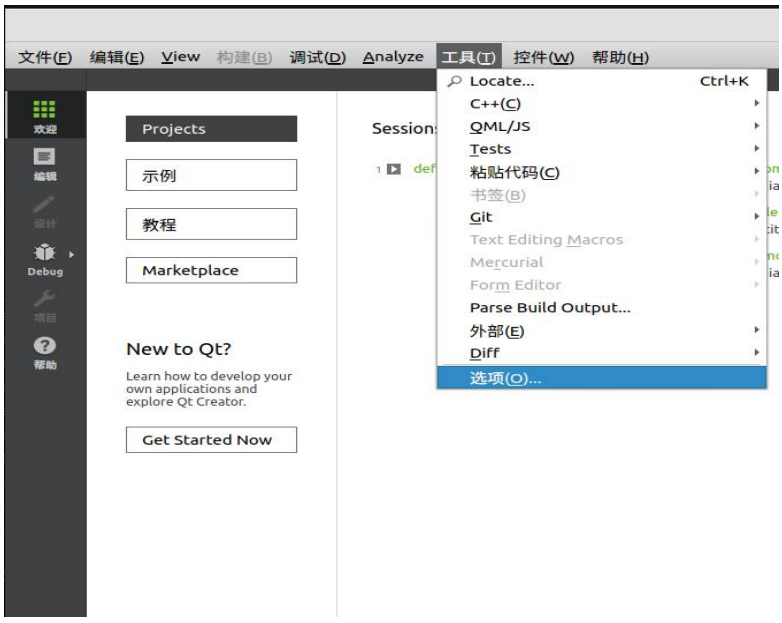
4.1 编译C/C++语言程序

```
1| #安装工具链
2| ./loongarch64-toolchain.sh
3| #设置环境变量
4| source /opt/poky/3.3+snapshot/environment-setup-
loongarch64-poky-linux
5| #编译
6| $CC -o hello hello.c
```

4.2 QT Creator交叉开发环境搭建

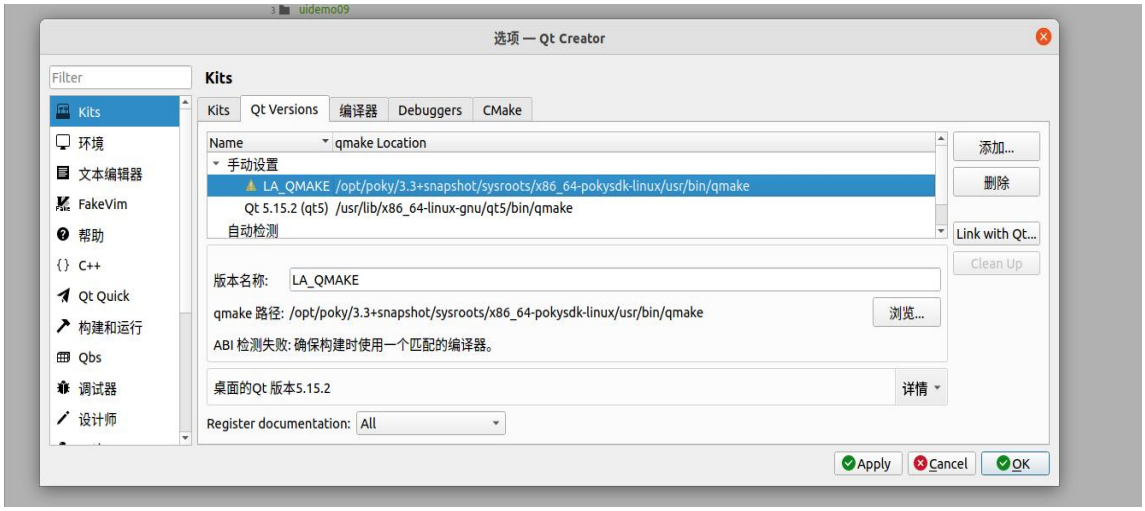
步骤如下：

1. 选择菜单栏的工具-选项



2. 设置qmake,如图Kits->QT versions 里添加qmke路径

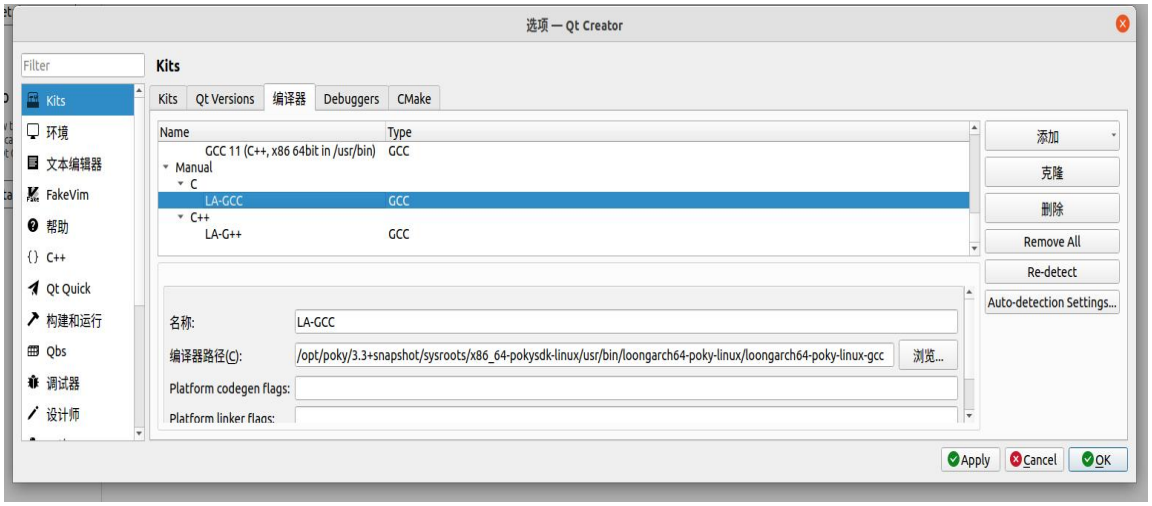
`/opt/poky/3.3+snapshot/sysroots/x86_64-pokysdk-linux/usr/bin/qmake`



3. 设置gcc和g++，如图Kits->编译器里添加gcc路径：

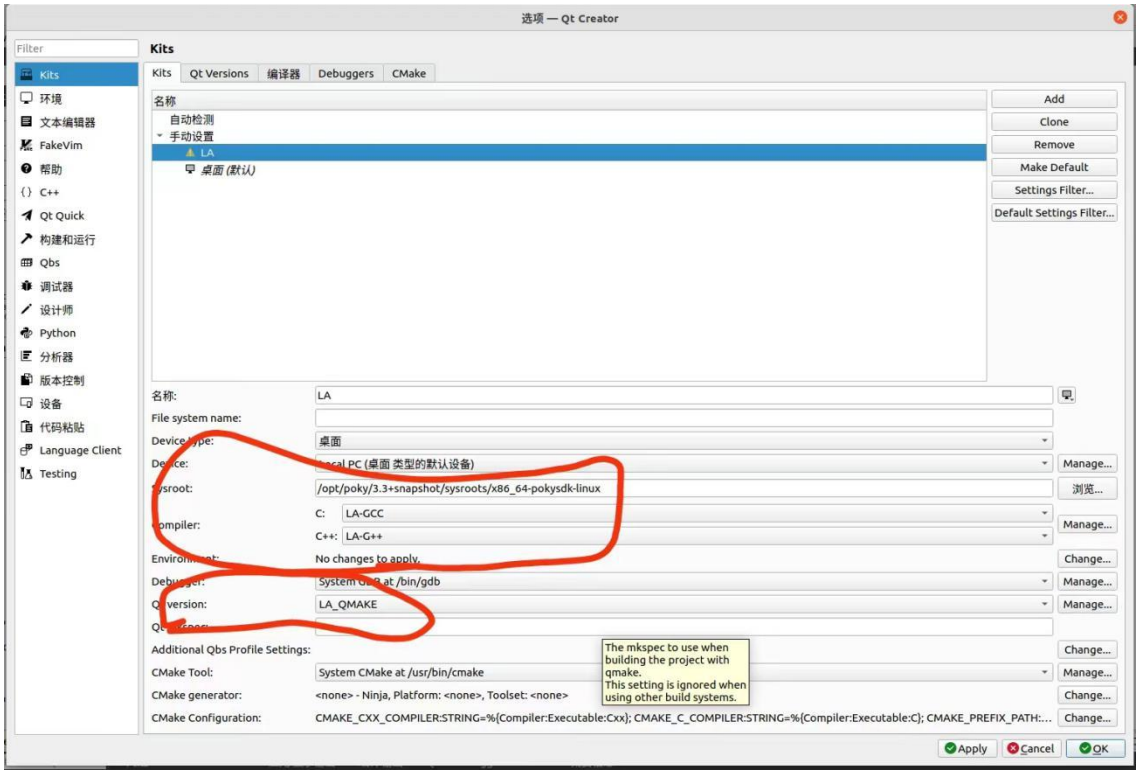
`/opt/poky/3.3+snapshot/sysroots/x86_64-pokysdk-linux/usr/bin/loongarch64-poky-linux/loongarch64-poky-linux-gcc`

4. g++路径：`/opt/poky/3.3+snapshot/sysroots/x86_64-pokysdk-linux/usr/bin/loongarch64-poky-linux/loongarch64-poky-linux-g++`



5. 设置Kits，如图Kits->Kits 里添加sysroot路径及上面设置的QMAKE、gcc、g++, sysroot路径：

`/opt/poky/3.3+snapshot/sysroots/x86_64-pokysdk-linux`



6. 设置完成后保存，编译工程时选择LA即可

4.3 编译QT程序

资料包提供了编译应用程序的交叉工具链，主要针对QT应用

下载loongarch64-toolchain.sh，并执行，具体流程如下：

```
1| ./loongarch64-toolchain.sh
2| source /opt/poky/3.3+snapshot/environment-setup-
loongarch64-poky-linux
3| #查看qmake版本
4| qmake -v
5| #进入需要编译的程序，命令行编译
6| qmake
7| make
```

5. 地址空间详解

5.1 pcie空间

龙芯2K1000的硬件资源大多都在pcie总线上，这些外设的基址也需要通过访问pcie的配置空间得到

type0访问内部资源，访问基址0xba000000,读取对应设备配置空间基址偏移0x10的地址，基址如下

设备	配置空间访问	默认值
apb	0xba001000	0x1fe20000
gmac0	0xba001800	0x40040000
gmac1	0xba001900	0x40050000
usb-otg	0xba002000	0x40000000
usb-ehci	0xba002100	0x40060000
usb-ohci	0xba002200	0x40070000
gpu	0xba002800	0x40080000
dc	0xba003000	0x400c0000
hda	0xba003800	0x400d0000
sata	0xba004000	0x400e0000
pcie0-p0	0xba004800	0x40100000
pcie0-p1	0xba005000	0x50000000
pcie0-p2	0xba005800	0x54000000
pcie0-p3	0xba006000	0x58000000
pcie1-p0	0xba006800	0x60000000
pcie1-p0	0xba007000	0x70000000
dma	0xba007800	

5.2 apb设备地址

uart\n	0x1fe20n00
can0	0x1fe20c00
can1	0x1fe20d00
i2c0	0x1fe21000
i2c1	0x1fe21800
pwm\n	0x1fe220n0
hpet	0x1fe24000

ac97	0x1fe25000
nand	0x1fe26000
acpi	0x1fe27000
rtc	0x1fe27800
des	0x1fe28000
aes	0x1fe29000
rsa	0x1fe2a000
rng	0x1fe2b000
sdio	0x1fe2c000
I2S	0x1fe2d000
E1	0x1fe2e000

type1访问外部pcie设备配置空间,基址0xbb000000,配置空间访问

31~24	23~16	15~11	10~8	7~0
0xbb	bus号	设备号	功能号	寄存器

5.3 spiflash

0~0xfb000	程序
0xfb000~0xff000	dtb
0xff000~0xff200	env
0xff200~0x100000	保留

6. 复用配置说明

GPIO与各接口复用关系见处理器手册GPIO章节13.5

6.1 PMON下配置复用

6.1.1 自动

Targets/ls2k/conf/ls2k为PMON配置文件，其中的 CONFIG_REG0/1/2分别对应处理器手册中的通用配置寄存器0/1/2。修改为需求数值即可完成复用，若没有使用PINCTRL，kernel下复用情况会与PMON下保持一致。

6.1.2 手动

`d4 0x1fe00420`查看复用

`m4 0x1fe00420 0x1f49`配置复用

6.2 kernel下配置复用

6.2.1 自动

使用PINCTRL子系统，内核选择配置PINCTR和PINCTRL_LS2K.

DTS中节点为

```
1|    pctrl:pinctrl@1fe00420 {
2|        compatible = "loongson,2k1000-pinctrl";
3|       reg = <0 0x1fe00420 0 0x18 >;
4|
5|        sdio_default:sdio {
6|                mux{
7|                groups = "sdio";
8|                function = "sdio";
9|                };
10|            };
11|         pwm0_default:pwm0 {
12|                mux {
13|                groups = "pwm0"
14|                function = "gpio"
15|                };
16|            };
17|    };
```

且对应设备数节点下要有pinctrl键值对，例如上述节点就需要在sdio节点中添加如下键值对

```
1| pinctrl-0 = <&sdio_default>;
2| pinctrl-names = "default";
```

6.2.1 手动

`devmem2 0x1fe00420`查看复用

`devmem2 0x1fe00420 w 0x1f49`配置复用

7. PMON常用指令汇总

7.1 h

功能：显示当前PMON全部可用命令

句式：h 或h 想查询的指令

用例：`h load` 详细显示该命令的句式结构与可用参数

7.2 ifconfig/ifaddr

功能： 设置对应网口IP，该设置不被保存

句式：ifconfig/ifaddr 网口名 IP号

用例：`ifconfig syn0 192.168.1.50 / ifaddr syn0 192.168.1.50`

注： PMON下使用该命令设置某网口IP后，若需临时使用另一网口，则需要`ifconfig syn0 remove` 来移除已经配置灯网口才能对新网口进行配置。

7.3 ping

功能：网络链路测试

句式： ping 主机IP

用例：`ping 192.168.1.10`

注：请确保网口设置成功IP后再ping

7.4 set

功能：查看或增改PMON环境变量

句式：set 或 set 环境变量名 环境变量值

用例：`set`无参数时会打印所有当前环境变量，回车继续，q退出。

`set ifconfig syn0:192.168.1.50` 添加环境变量ifconfig其值为syn0:192.168.1.50，再次启动时对应的网口自动设置IP.

`set al1 (wd0,0)/boot/vmlinuz`启动后自动加载该路径二进制文件作为内核文件，并启动内核。

注： 请谨慎修改现有环境变量

7.5 unset

功能：删除PMON环境变量

句式：unset 环境变量名

用例：`unset ifconfig`

注： 请谨慎修改现有环境变量

7.6 load

功能：load 命令是 PMON 中一个很重要命令。作用是加载一个 elf 文件到内存中(这里只是存放到内存中,而没有烧写到 flash 中)，加载过程是一个自动根据 elf 文件的信息处理elf 文件重定向的等等操作的总体过程,所以这里不需要指定加载的内存地址,load 命令会自动完成。如果指定加载的文件不是 elf 文件,将提示错误。

句式：load 所在路径/文件名称

用例：以下分别为U盘加载，硬盘加载，NVME硬盘加载，网络加载，nand加载

`load (usb0,0)/boot/vmlinuz` 将U盘的首个分区中的boot目录下的vmlinuz文件并作为内核加载

`load (wd0,0)/boot/vmlinuz`

`load (nvme0,0)/boot/vmlinuz`

`load tftp://192.168.1.11/vmlinuz`

`load /dev/mtd` 查看当前系统下所拥有的可操作flash

`load /dev/mtd1` 读取mtd1中数据作为内核二进制

7.7 initrd

功能：加载ramdisk作为文件系统。

句式：initrd 所在路径/文件名称

用例：以下分别为U盘加载，硬盘加载，NVME硬盘加载，网络加载

`initrd (usb0,0)/boot/rootfs.cpio.gz` 将U盘的首个分区中的boot目录下的rootfs.cpio.gz文件并作为文件系统加载

```
initrd (wd0,0)/boot/rootfs.cpio.gz
initrd (nvme0,0)/boot/rootfs.cpio.gz
initrd tftp://192.168.1.11/rootfs.cpio.gz
```

7.8 g

功能：g 命令是 PMON 中一个很重要的命令,直接从指定内存地址处开始执行程序

句式：g 内核传参

用例：`g console=ttyS0,115200`

内核加载完成后,使用 g 命令自动从加载后的内核入口址开始执行内核,这是 g 后面在参数 “console=ttyS0,115200 ”是内核启动的数。

7.9 fload

功能： 固件烧录

句式：fload 所在路径/文件名称

用例：以下分别为U盘加载，硬盘加载，NVME硬盘加载，网络加载

```
fload (usb0,0)/gzrom-dtb.bin
fload (wd0,0)/gzrom-dtb.bin
fload (nvme0,0)/gzrom-dtb.bin
fload tftp://192.168.1.11/gzrom-dtb.bin
```

注： 烧录过程会先擦除原始固件请确保整个过程不要中断或断电，当重新回到“PMON>”命令行模式则烧录完成，可以进行重启。

7.10 bl

功能：跳转运行某功能，暂时只用于手动读取boot.cfg

句式：bl 所在路径/boot.cfg

用例：以下分别为U盘加载，硬盘加载，NVME硬盘加载，网络加载

```
bl -d ide (usb0,0)/boot/boot.cfg
bl -d ide (wd0,0)/boot/boot.cfg
bl -d ide (nvme0,0)/boot/boot.cfg
bl -d ide tftp://192.168.1.11/boot.cfg
```

7.11 fdt(dtb相关)

功能：设备树操作工具集

句式：请从`fdt`打印信息了解各工具句式

用例：`fdt` 查看当前可用设备树操作命令

打印所有dtb

```
print_dtb /
```

打印dc节点

```
print_dtb /soc/dc@0x400c0000
```

修改mac地址

```
set_dtb /soc/ethernet@0x40040000 mac-address [80 c1 80 c1
80 c1]
```

更新dtb

```
load_dtb tftp://xx.xx.xx.xx/ls2k.dtb
```

擦除dtb

```
erase_dtb
```

7.12 m1/m2/m4/m8

功能：向给定地址写入对应长度的数据

句式：m4 地址 写入内容

用例：`m4 0x1fe00420 0x00ff0d43` 向0x1fe00420地址写入四字节数据0x00ff0d43

注：请勿随意写入，各地址各位功能参考所用芯片的处理器用户手册

7.13 d1/d2/d4/d8

功能：从给定地址读取对应长度的数据

句式：d4 地址 读取次数

用例：`d4 0x1fe00420` 从0x1fe00420读取四字节数据

`d4 0x1fe00420 4` 一次4字节的读取0x1fe00420到0x1fe00430的数据

7.14 devls

功能：显示当前PMON下检测到的可用设备

句式：devls

用例：`devls`

7.15 devcp

功能：打开 src-device 和 desc-device 两个设备，从 src-device 读取一定数量的字节数据，写到 desc-device 设备中去，完成后关闭这两个设备。可以理解为将目标内容拷贝入目标空间。

句式：devcp src-device desc-device

用例：`devcp tftp://server-ip/vmlinux /dev/mtd0`

7.16 mtd_erase

功能：擦除 nandflash 的一个分区

句式：mtd_erase desc-device

用例：

`mtd_erase /dev/mtd0`

(擦除 nandflash 的第一个分区，跳过已经是坏块的地方，不会尝试擦除已经是坏块的地方)

`mtd_erase /dev/mtd0r`

(擦除 nandflash 的第一个分区,这时会尝试擦除所有的块,即使是坏块也会尝试执行擦除操作)

7.17 pciscan

功能：扫描并输出PCI总线上的所有设备

句式：pciscan

用例：`pciscan`

7.18 data

功能：读写内部RTC

句式：data / date [yyyymmddhhmm.ss]

用例：`date` 读取当前时间

`date 202211010955.30` 设置当前时间

7.19 pcs

功能：选定PCI设备功能

句式：pcs 功能选项

用例：`pcs -1`

注：-1：64位uncached物理地址

-2：64位cached物理地址

-3：64位CPU地址

-4：32位CPU地址

7.20 spi_base

功能：spi操作片选，默认为0

句式：spi_base 片选

用例：`spi_base 0`

7.21 spi_id

功能：读取当前片选下flashID

句式：spi_id

用例：`spi_id`

7.22 erase_all

功能：擦除当前操作的SPI flash

句式：erase_all

用例：`erase_all`

7.23 read_pmon

功能：读取flash内容

句式：read_pmon 开始地址 读取大小

用例：`read_pmon 0 0x100`

7.24 eepread

功能：一次一字节读取EEPROM芯片数据

句式：eepread 开始地址 读取数量

用例：`eepread 0 10`

7.25 eepwrite

功能：按字节写入EEPROM

句式：eepread 开始地址 “一字节数据 一字节数据 ....”

用例：`eepread 0x5 “12 34 56”`

7.26 lwdhcp

功能：使用DHCP自动分配IP

句式：lwdhcp 网口名

用例：`lwdhcp` 默认自动分配syn0地址

`lwdhcp syn1` 自动分配syn1地址

7.27 vers

功能：查看当前PMON版本

句式：vers

用例：`vers`

附录：FAQ

1、pmon编译报错,pmoncfg:not found。

参考方案：

编译时未找到pmoncfg,在pmon顶层目录中tools/pmoncfg/目录下执行make,成功生成pmoncfg后再编译PMON。

2、pmon编译时dtb编译报错如同

```
lspai/pmon-ls2k1000la/zloader.ls2k/../Targets/ls2k/conf/ls2k.dts
make[1]: 离开目录“/home/linux/loongson/la/2k1000la/lspai/pmon-ls2k1000la/Targets/ls2k/compile/ls2k”
./dtc -I dts -O dtb -o ls2k.dtb  ls2k.dtb.i
Error: /home/linux/loongson/la/2k1000la/lspai/pmon-ls2k1000la/zloader.ls2k/../Targets/ls2k/conf/ls2k.dts:795.3-796.1 syntax error
FATAL ERROR: Unable to parse input tree
Makefile.inc:84: recipe for target 'dtb' failed
make: *** [dtb] Error 1
```

参考方案：

使用的DTS中有格式错误，一般为少了顿号或大括号。

3、pmon编译时DTB编译报错如同

```
make[1]: 离开目录“/home/linux/loongson/la/2k1000la/lspai/p
./dtc -I dts -O dtb -o ls2k.dtb  ls2k.dtb.i
make: execvp: ./dtc: 权限不够
Makefile.inc:84: recipe for target 'dtb' failed
make: *** [dtb] Error 127
```

参考方案：

可能是在WINDONS中解压了源码包导致文件权限异常，建议在LINUX环境下重新解压一份源码。

4、进入EJTAG操作时出现如下打印

参考方案：这种情况一般为使用虚拟机的用户，请重启虚拟机后再测试。

5、烧录PMON时按照手册操作后set依然无相应

参考方案：

（1）检查EJTAG连接是否正确，连接器上标三角的为1号引脚，若使用小接口EJATG连接器，则丝印上有三角的为1号引脚。

（2）检查所连接的板上EJTAG接口是否为烧录pmon使用，有些板子上有多个EJTAG口，一般丝印为CPU_EJTAG的为烧录PMON使用的。

（3）检查所使用的EJTAG烧录工具包是否过旧，若早于22年9月请跟换最新烧录工具包使用

（4）检查CPU是否正常上电，板路走线是否连通正常，上电并执行过`source configs/config.ls2k`语句后使用`jtagregs d8 1 1`来确定ejtag是否于cpu连接正常若连接正常则应如下打印

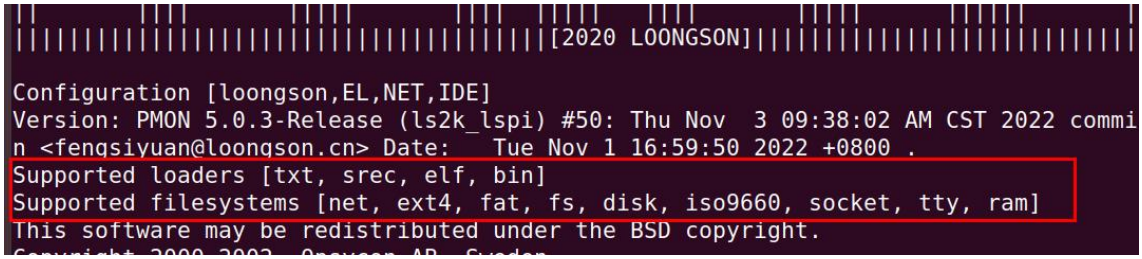
```
cpu0 -jtagregs d8 1 1
00000001: 000000005a5a5a5a                    ZZZZ....
```

（5）若上述方案失败，可在进入EJTAG后使用`jtag_clk 8`命令来降低速率，之后再运行`jtagregs d8 1 1`测试。降速选项有2/4/8数值越大速率约低。

6、PMON下支持的文件格式

参考方案：

U盘建议使用FAT硬盘建议使用EXT4，其他支持的引导文件格式与文件系统格式请看PMON启动打印



7、板卡上电后，串口无打印。

参考方案：

请确认使用的串口是否为调试串口，没有RS232接口的板卡需要注意是否将读写引脚反接。若之前有进行过对SPI片选0的擦除，请重新烧写PMON。

8、板卡上电后，串口打印乱码。

参考方案：

（1）检查调试工具串口设置是否为速率115200，类型8N1.

（2）没有RS232接口的板卡，需要确认地线是否有虚接。

（3）板卡调试串口信号为232，请使用232串口线，若使用其余串口测试则需要确认其输出信号是TTL还是232以采用对应转换调试线。

（4）虚拟机用户，在修改内存频率后有概率出现乱码，请断开串口线与虚拟机的连接后重启虚拟机再做测试。

9、需要使用GPIO中断功能，有例程吗？

参考方案：

gpio中断的测试可以参考2.8章节中测试驱动来测试，当前2K1000LA只有GPIO0-3可以设备为高/低电平或上/下边沿触发，GPIO4-31与GPIO32-64因使用共享中断所以只可使用高电平触发中断，或共享中断GPIO组中只使用其中一个GPIO来注册中断则上述四种触发也可用。推荐按照2.8章节中gpio_keys_polled驱动来代替中断方案。

10、/dev/ttySx是对应的哪一个UART。

参考方案：

可以查看设备树如下节点内容，serialx对应ttySx.

```
aliases {
        ethernet0 = &gmac0;
        ethernet1 = &gmac1;
        serial0 = &cpu_uart0;
        serial1 = &cpu_uart1;
        serial2 = &cpu_uart2;
        serial3 = &cpu_uart3;
        i2c0 = &i2c0;
        i2c1 = &i2c1;
};
```

11、内核下SPI可以操作的片选怎么看。

内核中/dev下节点spidev0,x为空闲片选，x的数字代表所对应的片选。

其与设备树中如下节点有关

```
spi0: spi@0x1fff0220{
    compatible = "loongson,ls-spi";
    #address-cells = <1>;
    #size-cells = <0>;
    reg = <0 0x1fff0220 0 0x10>;
    spidev@0{
        compatible = "jedec,spi-nor";
        spi-max-frequency = <40000000>;
        reg = <0>;
    };
    spidev@1{
        compatible = "rohm,dh2228fv";
        spi-max-frequency = <40000000>;
        reg = <1>;
    };
};
```

红框所标出的部分为会显示在/dev下的空闲节点。

12、/dev/mtdn分别与设备是怎样的对应关系。

PMON下对应情况使用load /dev/mtd来查看，如下图

```
PMON> load /dev/mtd
mtd0: flash:nand-flash type:nand size:0x20000000 writesize:0x800 erasesize:0x20000 partoffset=0x0,partsize=0x1e00000 kernel
mtd1: flash:nand-flash type:nand size:0x20000000 writesize:0x800 erasesize:0x20000 partoffset=0x1e00000,partsize=0x1e0e0000 rootfs
mtd2: flash:spinand_flash type:nand size:0x20000000 writesize:0x400 erasesize:0x10000 partoffset=0x0,partsize=0x13b0000 kernel
mtd3: flash:spinand_flash type:nand size:0x20000000 writesize:0x400 erasesize:0x10000 partoffset=0x1400000,partsize=0x1eb90000 os
mtd4: flash:m25p800 type:nor size:0x1000000 writesize:0x1 erasesize:0x1000 total
mtd5: flash:m25p800 type:nor size:0x1000000 writesize:0x1 erasesize:0x1000 partoffset=0x0,partsize=0x1400000 kernel
mtd6: flash:m25p800 type:nor size:0x1000000 writesize:0x1 erasesize:0x1000 partoffset=0x1400000,partsize=0xfffffffffc00000 os
/dev/mtd: Undefined error: 0
PMON>
```

内核下对应情况可用使用cat /proc/mtd来查看

```
[root@buildroot ~]# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 01400000 00020000 "nand_kernel_partition"
mtd1: 1ec00000 00020000 "nand_os_partition"
mtd2: 01400000 00020000 "spinand_kernel_partition"
mtd3: 1ec00000 00020000 "spinand_os_partition"
[root@buildroot ~]#
```

13、启动时内核打印到init=/sbin/init后弹栈或卡死

可能为所使用的文件系统的初始化文件错误，可以实验将内核传参中的rdinit改为

rdinit=/init或直接删去rdinit

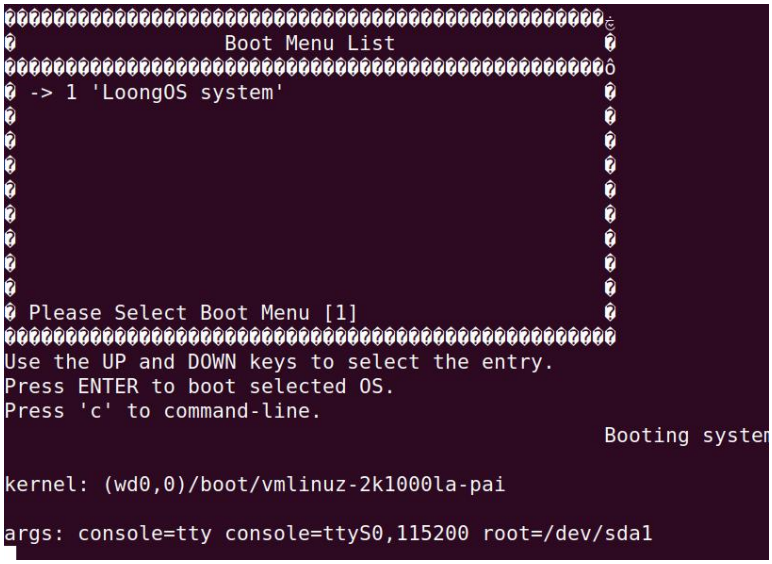
14、U-BOOT可用功能现状

当前U-BOOT为V0.1版本，该版本可通过硬盘或网络TFTP加载的方式启动内核，当前仅适配LA龙芯派平台，部分功能性驱动在U-BOOT下暂不可用。

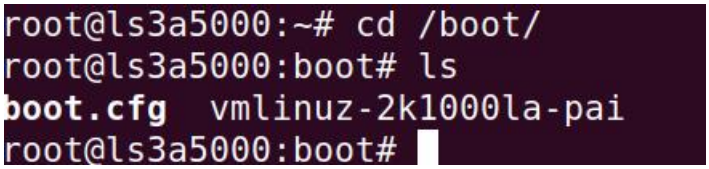
15、使用硬盘挂载方式的系统如何替换内核

以龙芯派板载LOONGOS系统更换内核为例

启动时在BOOT MENU LIST倒计时中，使用“↑”或“↓”方向键可以停止倒计时并且显示“->”对应选项所代表的启动参数，如下图



可以看到该选项所对应的内核位于硬盘首个分区中boot目录下，此时我们先按回车进入系统。可以在/boot/中找到对应的内核文件如图



只需通过U盘或网络加载并替换其中的内核二进制即可完成内核替换。若替换前后的名称不同则只需修改该目录下的boot.cfg中对应的内核二进制名称即可。

16、使用硬盘挂载方式的系统如何替换硬盘中的文件系统

建议使用网络或U盘加载的方式，加载的文件系统可用要换入的新系统，加载的内核可以自己使用cpio命令展开新系统的cpio包后从boot中获得，也可从旧系统的/boot目录中考出使用。进入内核后，手动挂载并清空系统硬盘分区，将要替换的文件系统拷贝入该分区并使用cpio命令展开。以LOONGOS为例，展开后的文件中boot目录下自带内核与boot.cfg文件，取消挂载后重启即可等待新系统自

启。

17、cpio.gz文件系统的展开与合包

需要向文件系统中加入某些共享库或可直接执行的脚本或二进制文件时，可在root用户下采用以下命令操作

```
gunzip xxx.cpio.gz     //解压
cpio -idmv < xx.cpio   //展开，建议在一个独立目录下进行
cp  xx  xxx                  //放入所需文件
find ./* | cpio -H newc -o > xx.cpio //合包
gzip xx.cpio           //压缩
```