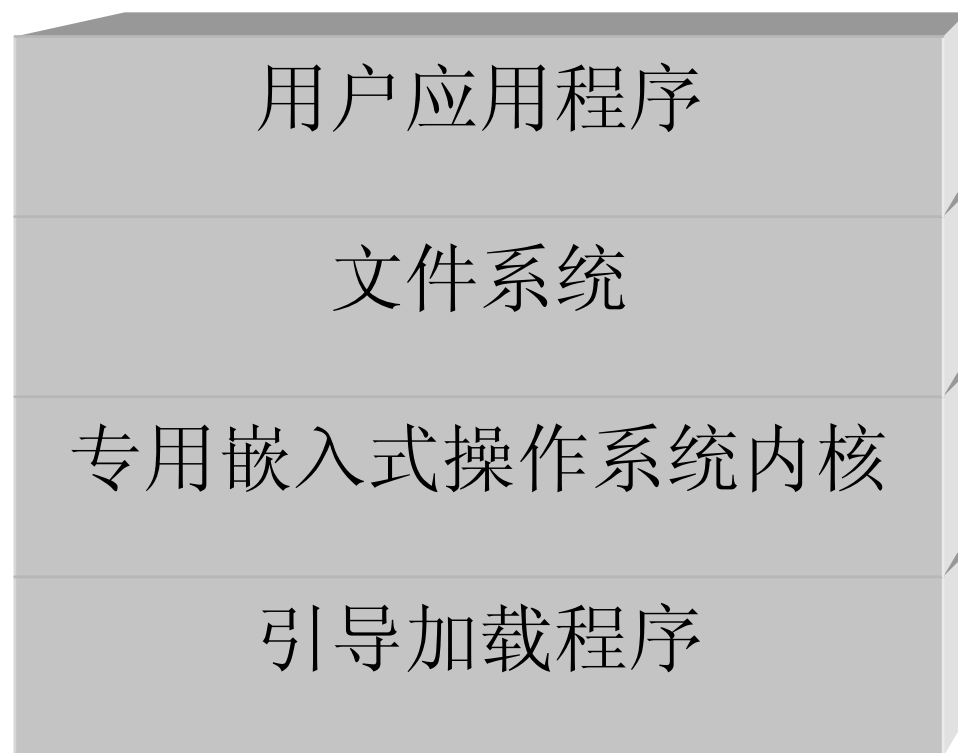


# PART I BOOTLOADER

## — 概述

# 嵌入式系统软件体系架构



# Bootloader概念

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

■ **Boot Loader** 是严重地依赖于硬件而实现的，特别是在嵌入式世界。因此，在嵌入式世界里建立一个通用的 **Boot Loader** 几乎是不可能的

# Bootloader概念

■ **Boot Loader 所支持的 CPU 和嵌入式板：** 每种不同的 CPU 体系结构都有不同的 Boot Loader。有些 Boot Loader 也支持多种体系结构的 CPU

■ **Boot Loader 的安装媒介：** 系统加电或复位后，所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。比如，基于 ARM7TDMI core 的 CPU 在复位时通常都从地址 0x00000000 取它的第一条指令

# Bootloader概念

- 用来控制 **Boot Loader** 的设备或机制：主机和目标机之间一般通过串口建立连接，**Boot Loader** 软件在执行时通常会通过串口来进行 I/O，比如：输出打印信息到串口，从串口读取用户控制字符
- **Boot Loader** 的操作模式："启动加载"模式和"下载"模式

# Bootloader概念

■ 启动加载（**Boot loading**）模式：这种模式也称为"自主"（**Autonomous**）模式。也即 **Boot Loader** 从目标机上的某个固态存储设备上将操作系统加载到 **RAM** 中运行，整个过程并没有用户的介入。

# Bootloader概念

■ **下载（Downloading）模式**：在这种模式下，目标机上的 **Boot Loader** 将通过串口连接或网络连接等通信手段从主机（**Host**）下载文件，从主机下载的文件通常首先被 **Boot Loader** 保存到目标机的 **RAM** 中，然后再被 **Boot Loader** 写到目标机上的 **FLASH** 类固态存储设备中。**Boot Loader** 的这种模式通常在第一次安装内核与根文件系统时被使用；此外，以后的系统更新也会使用 **Boot Loader** 的这种工作模式。工作于这种模式下的 **Boot Loader** 通常都会向它的终端用户提供一个简单的命令行接口。

# Linux的开放源码Boot Loader 以及其所支持的架构

Boot Loader	监控程序	说明	架构					
			X86*	ARM	PowerPC	MIPS	M68k	SuperH
LILO	否	Linux主要的磁盘引导加载程序	*					
GRUB	否	LILO的GNU版后继者	*					
ROLO	否	不需要BIOS可直接从ROM加载Linux	*					
Loadlin	否	从DOS加载Linux	*					



# Linux的开放源码Boot Loader 以及其所支持的架构

Etherboot	否	从ETHERNET卡启动系统的Romable loader	*					
LinuxBIOS	否	以Linux为基础的BIOS替代品	*					
Compaq的bootldr	是	主要用于Compaq iPAQ的多功能加载程序		*				

# Linux的开放源码Boot Loader 以及其所支持的架构

blob	否	来自LART硬件计划的加载程序		*				
PMO N	是	Agenda VR3中所使用的加载程序				*		
sh- boot	否	LinuxSH计划的主要加载程序						*
U- Boot	是	以PPCBoot和ARMBoot为基础的通用加载程序	*	*	*			
Red Boot	是	以eCos为基础的加载程序	*	*	*	*	*	*

# Bootloader作用

- 设计程序入口指针
- 建立异常中断处理向量
- 初始化**CPU**各种模式的堆栈和寄存器
- 初始化系统中要使用的各种片内外设
- 初始化目标板
- 引导操作系统。

# PART II 典型嵌入式bootloader

—vivi

# vivi简介

- vivi 是韩国Mizi 公司开发的BootLoader，适用于ARM9 处理器。
- vivi 有两种工作模式，启动加载模式可以在一段时间后（这个时间可更改）自行启动Linux 内核，这是vivi的默认模式。
- 在下载模式下，vivi 为用户提供一个命令行接口，通过该接口可以使用vivi提供的一些命令

# vivi命令

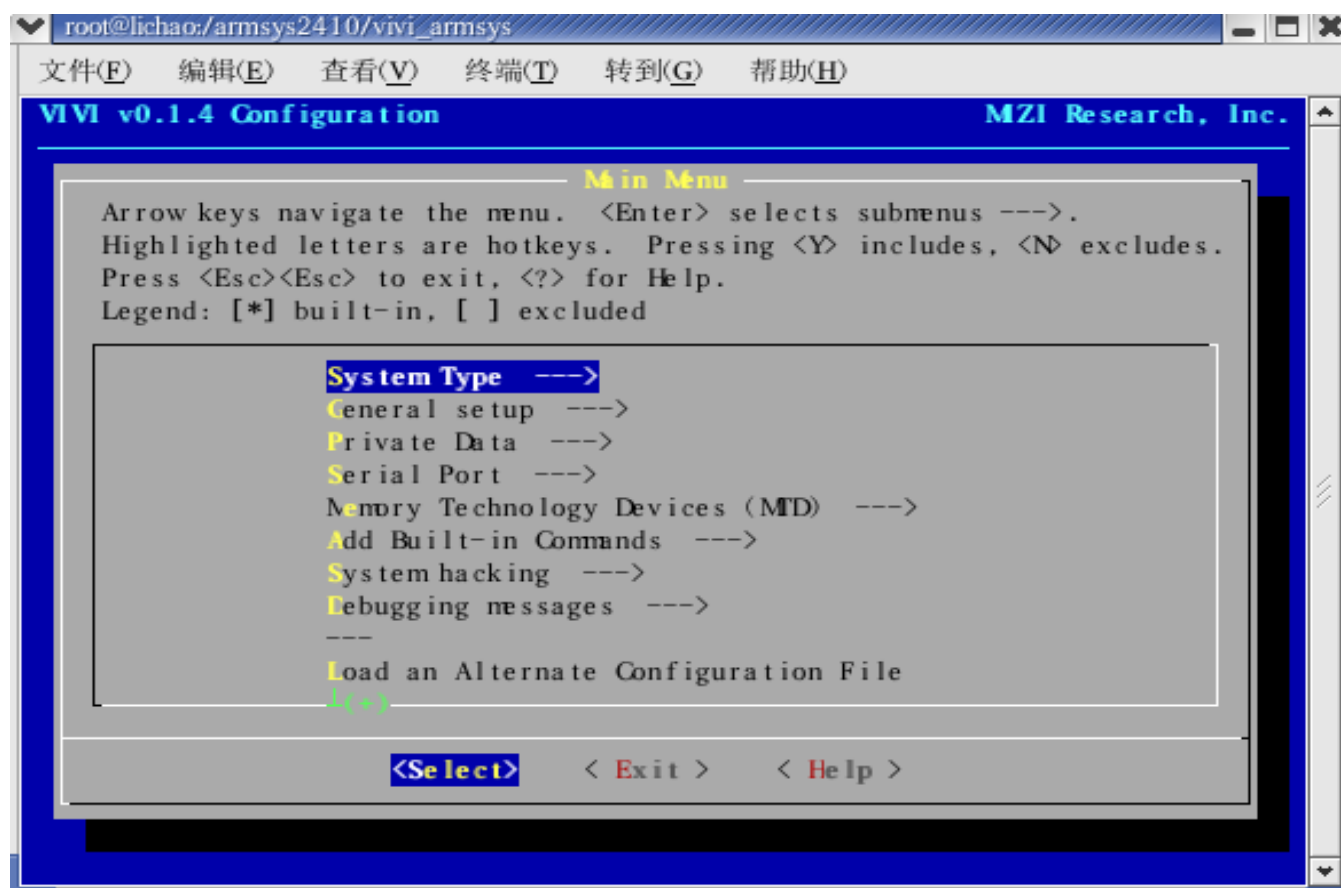
命令	功能
load	把二进制文件载入Flash 或者RAM
part	操作MTD 分区信息，显示、增加、删除、复位、保存MTD 分区
param	设置参数
boot	启动参数
flash	管理Flash，如删除Flash 数据

## vivi安装

- (1)cd /
- (2) mkdir armsys2410
- (3)将vivi\_armsys.tgz拷贝到armsys2410目录下
- (4)tar xzvf vivi\_armsys.tgz
- (5)cd vivi\_armsys

# vivi裁減

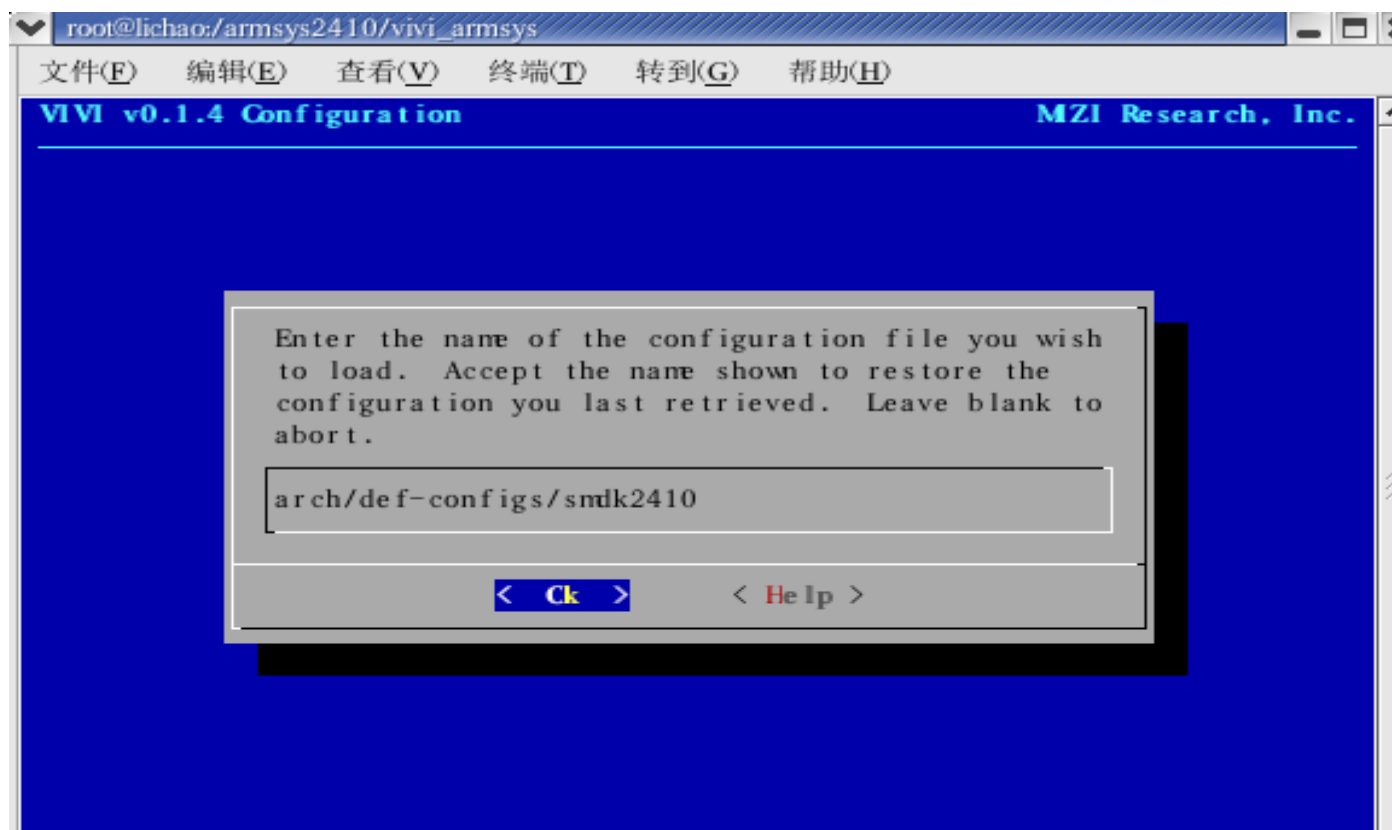
- (1) make distclean
- (2) make menuconfig





# vivi裁減

- (1)选择“ Load on Alternate Configuration File ” 菜单
- 然后写入arch/def-configs/smdk2410



# vivi编译

- make
- 在vivi\_armsys目录下生成vivi二进制文件
- vivi.map文件

# vivi固化到开发板上

- `mkdir /armsys2410/Jflash`
- `cd Jflash`
- 将jflash-s3c2410文件和vivi拷贝到Jflash目录
- `./jflash-s3c2410 vivi /t=5`

[illegible]

# PART III 典型嵌入式bootloader

## —vivi使用

## part 命令

- `part show` 显示分区信息
- `part add partname part_start_addr part_leng 0` 添加分区
- `part del partname` 删除分区
- `part save` 保存part 分区信息

# load 命令

- `load flash partname x` 使用xmodom 协议通过串口下载文件并且烧写带partname 分区
  - `load flash vivi x`
  - `load flash kernel x`
  - `load flash root x`
- `load ram partname or addr x` 使用xmodom 协议通过串口下载文件到内存中

# param命令

- param 命令
- param show 显示配置信息
- param set paramname value 设置参数值
- param set linux\_cmd\_line "linux boot param" 设置linux 启动参数
- param save 保存参数的设置



# boot命令

- `boot boot linux` 操作系统
- `boot ram ramaddr lenth` 启动以及下载到  
s dram 中的linux 内核。

# bon 命令

例如分为3个区： 0~192k, 192k~1M, 1M~

**vivi> bon part 0 192k 1M**

**doing partition**

**size = 0**

**size = 196608**

**size = 1048576**

**check bad block**

**part = 0 end = 196608**

**...**

## go 命令





















- `go addr` 跳转到指定地址运行该处程序。

# PART IV 典型嵌入式bootloader

## —vivi代码体系架构分析

# vivi代码体系架构分析

- vivi是一个源代码开放的bootloader，并且它的代码组织形式非常类似于linux，因此熟悉linux源代码结构的读者，会比较容易理解vivi代码的构造。
- 解压后的vivi代码大概有200个左右的文件构成，分布在arch，init，drivers，lib等几个目录下

-  vivibootloader
  -  vivi
    -  arch
      -  cvs
      - +  def-configs
      - +  s3c2410
      -  cvs
      - +  Documentation
      -  drivers
        -  cvs
        - +  mtd
        - +  serial
      - +  include
      - +  init
      -  lib
        -  cvs
        - +  priv\_data
      - +  scripts
      - +  test
      - +  util

# arch 目录

- arch 目录是vivi代码体系架构中最重要的目录之一，该目录下存放着和体系架构相关的源码
- arch目录下包含两个子目录，分别为def-configs和S3C2410，其中def-configs中存放了系统缺省的配置文件，相当于Linux进行配置后得到的.config文件

# arch 目录

- S3C2410目录包含的文件
  - head.S文件是VIVI启动后第一个执行的文件，该文件为汇编语言编写，完成系统的初始化阶段部分任务。
  - mmu.c文件是用来初始化MMU部件，它并不能实现全部的MMU功能，仅能完成简单的内存映射。
  - nand\_read.c文件中仅提供了一个函数实现，该函数完成从NAND设备启动时读取数据的功能。
  - smdk.c文件中定义了嵌入式开发板相关的设置值，比如开发板内存分布情况。












存放vivi ( )	0x00000000
存放启动参数	0x0001FFFF 0x00020000
存放Linux内核	0x0002FFFF 0x00030000
存放文件系统	0x001FFFFFFF 0x00200000
	0x01FFFFFFF

```
mtd_partition_t default_mtd_partitions[] = {
    {   name:           "vivi",
        offset:         0,
        size:           0x00020000,
        flag:           0
    }, {   name:         "param",
        offset:         0x00020000,
        size:           0x00010000,
        flag:           0
    }, {   name:         "kernel",
        offset:         0x00030000,
        size:           0x001d0000,
        flag:           0
    }, {
        name:           "root",
        offset:         0x00200000,
        size:           0x02000000,
        flag:           MF_BONFS
    }
};
```

# drivers 目录

- drivers 目录下存放着系统板驱动源码，包括二类设备的驱动代码：串口和MTD设备。
- 串口目录下的文件

---

	Config.in	1 KB	IN 文件
	getcmd.c	2 KB	C Source file
	getcmd.h	1 KB	C Header file
	getcmd_ext.c	7 KB	C Source file
	Makefile	1 KB	文件
	serial_core.c	2 KB	C Source file
	term.c	2 KB	C Source file
	xmodem.c	5 KB	C Source file
	ymodem.c	8 KB	C Source file

# init 目录

- init 目录下有 main.c 文件和 version.c 两个文件。其中 main.c 文件中有一个 main（）函数，它是 vivi 启动后第一个执行到的 C 函数（注意 vivi 启动后第一个执行的文件是 head.S）。

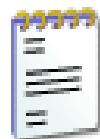
# lib目录



priv\_data



Config\_cmd.in  
IN 文件  
1 KB



heap.c  
C source file  
3 KB



memory.c  
C source file  
12 KB



printk.c  
C source file  
9 KB



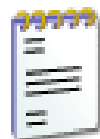
time.c  
C source file  
3 KB



boot\_kernel.c  
C source file  
11 KB



ctype.c  
C source file  
2 KB



load\_file.c  
C source file  
5 KB



memtst.c  
C source file  
11 KB



reset\_handle.c  
C source file  
2 KB



command.c  
C source file  
8 KB



exec.c  
C source file  
4 KB



Makefile  
文件  
1 KB



misc.c  
C source file  
4 KB



string.c  
C source file  
4 KB

# include 目录

- 该目录具有非常重要的作用，系统平台所使用到的头文件都存放在该目录下。其中 `s3c2410.h` 定义了 S3C2410 处理器中所使用到的寄存器， `platform/smdk2410.h` 定义了与此开发板相关的资源配置参数。

# PART V 典型嵌入式bootloader

## —vivi配置系统分析

# vivi配置系统结构

❖ vivi源码的配置系统由三个部分组成，分别是：

■ Makefile：分布在vivi源代码中的 Makefile，定义vivi内核的编译规则；

■ 配置文件（config.in）：给用户提供配置选择的功能；

■ 配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面、基于 Ncurses 图形界面以及基于 Xwindows 图形界面的用户配置界面，各自对应于 Make config、Make menuconfig 和 make xconfig）。



# vivi中的Makefile

- vivi源代码组织和Linux内核源代码组织比较类似，都是按照树形结构组织的，在每个目录下都会有一个Makefile文件，同时在最顶层目录下还有一个总纲领式的Makefile文件。所有这些Makefile文件的作用是根据配置的情况，构造出需要编译的源文件列表，然后分别编译，并把目标代码链接到一起，最终形成一个二进制可执行文件。

# vivi中与makefile相关的文件

- ■ Makefile: 顶层 Makefile, 是整个内核配置、编译的总体控制文件。
- ■ .config: vivi配置文件, 包含由用户选择的配置选项, 用来存放vivi配置后的结果。
- ■ 各个子目录下的 Makefile: 比如 drivers/Makefile, 负责所在子目录下源代码的管理。
- ■ Rules.make: 规则文件, 被所有的 Makefile 使用。

# vivi生成过程

- 用户通过 `make menuconfig` 配置后，产生了 `.config`。
- 顶层 `Makefile` 读入 `.config` 中的配置选择。
- 在vivi中，顶层 `Makefile`的任务只有一个，产生vivi目标文件，这可以从顶层 `Makefile`文件来进行解读。

# 顶层Makefile文件内容:

```
all:      do-it-all
```

```
.....
```

```
do-it-all:  Version vivi
```

```
.....
```

```
Version: dummy
```

```
    @rm -f include/compile.h
```

# 顶层Makefile文件内容:

```
vivi:include/version.h      $(CONFIGURATION)
    init/main.o init/version.o linuxsubdirs
    $(LD) -v $(LINKFLAGS) \
        $(HEAD) \
        $(CORE_FILES) \
        $(DRIVERS) \
        $(LIBS) \
        -o vivi-elf $(CLIBS)
    $(NM) -v -l vivi-elf > vivi.map
    $(OBJCOPY) -O binary -S vivi-elf vivi
    $(OBJCOPYFLAGS)

.....

include Rules.make
```

# 顶层makefile解读—1

- 为了生成vivi目标文件，顶层 Makefile 递归的进入到内核的各个子目录中，分别调用位于这些子目录中的 Makefile。至于到底进入哪些子目录，取决于内核的配置。
- 位于各个子目录下的 Makefile 同样也根据 .config 给出的配置信息，构造出当前配置下需要的源文件列表，并在文件的最后有 include \$(TOPDIR)/Rules.make。
- Rules.make 文件起着非常重要的作用，它定义了所有 Makefile 共用的编译规则。比如，如果需要将本目录下所有的 c 程序编译成汇编代码，需要在 Makefile 中有以下的编译规则：

```
%.s: %.c:
```

```
$(CC) $(CFLAGS) -S $< -o $@
```

# 顶层makefile解读—2

- `CONFIGURATION = config`
- `LD = $(CROSS_COMPILE)ld`
- `HEAD:= arch/$(MACHINE)/head.o`
- `CORE_FILES = init/main.o  
init/version.o lib/lib.o`
- `LIBS := lib/priv_data/priv_data.o`
- `SUBDIRS = drivers lib`

# 顶层makefile解读—3

1.

```
/usr/local/arm/2.95.3/bin/arm-linux-ld -v -Tarch/vivi.lds -Bstatic \  
    arch/s3c2440/head.o \  
    arch/s3c2440/s3c2440.o init/main.o init/version.o lib/lib.o \  
    drivers/serial/serial.o drivers/mtd/mtd.o \  
    lib/priv_data/priv_data.o \  
-o vivi-elf -L/usr/local/arm/2.95.3/lib/gcc-lib/arm-linux/2.95.3 -  
lgcc -lc
```

2.

```
/usr/local/arm/2.95.3/bin/arm-linux-nm -v -l vivi-elf > vivi.map
```

3.

```
/usr/local/arm/2.95.3/bin/arm-linux-objcopy -O binary -S vivi-elf vivi  
-R .comment -R .stab -R .stabstr
```



# vivi中的配置文件

- 除了 Makefile 的编写，另外一个重要的工作就是把新功能加入到vivi的配置选项中，提供此项功能的说明，让用户有机会选择此项功能。所有的这些都需要在 config.in 文件中用配置语言来编写配置脚本，在 vivi 内核中，配置命令可以有多种方式

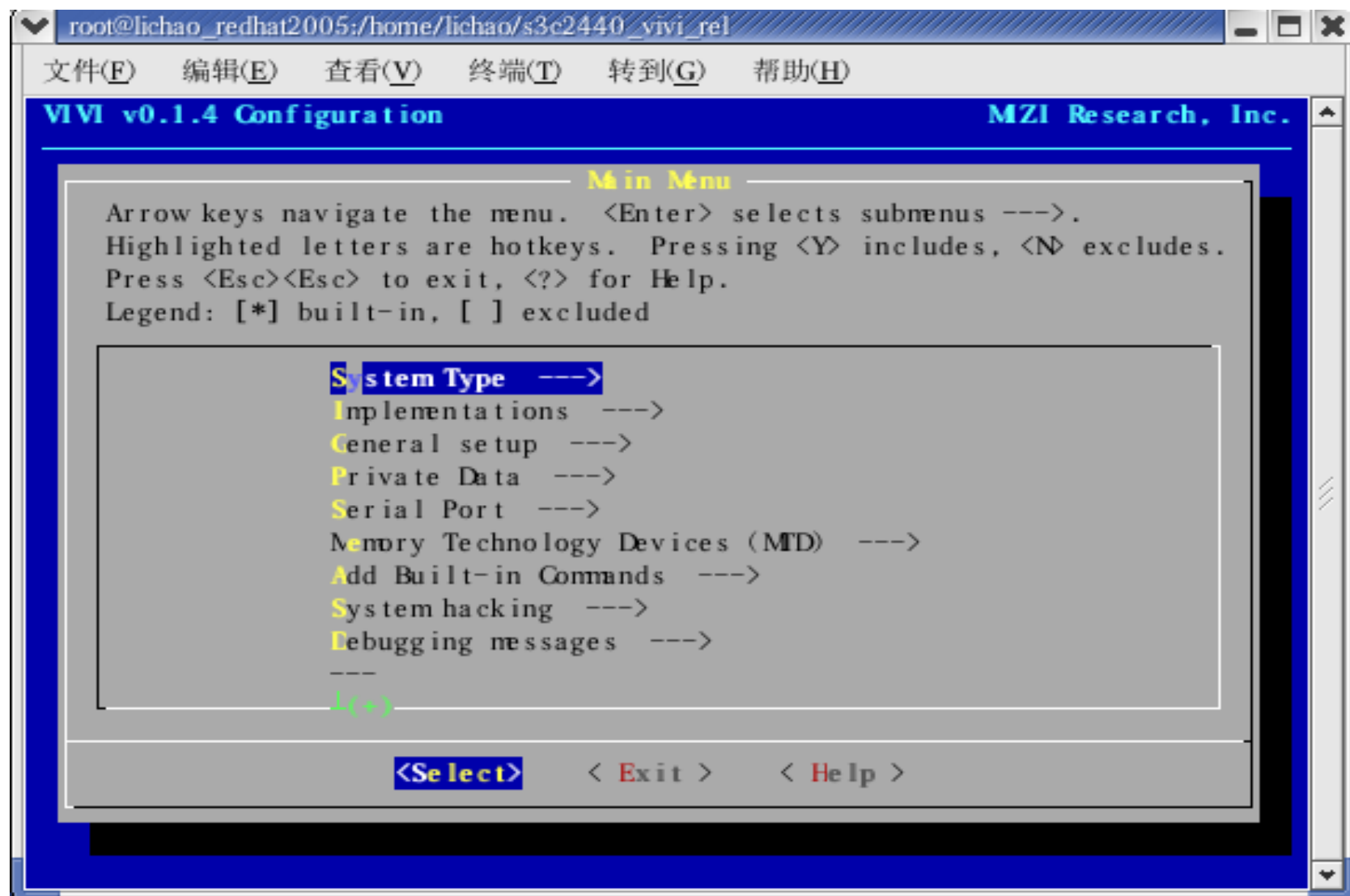
配置命令	解释脚本
make config make oldconfig	scripts/Configure
Make menuconfig	scripts/Menuconfig
Make xconfig	scripts/tkparse

# 字符配置界面—make config

```
[root@lichao_redhat2005 s3c2440_vivi_rel]# make config
/bin/sh scripts/Configure arch/config.in
#
# Using defaults found in .config
#
*
* System Type
*
ARM system type (SA1100-based, PXA250/210-based, S3C2400-based, S3C2410-based, S3C2440-based) [S3C2440-based] S
3C2440-based
    defined CONFIG_ARCH_S3C2440
*
* Implementations
*
Platform (SMDK, MPORT3) [SMDK] SMDK
    defined CONFIG_S3C2440_SMDK
    Support NAND Boot (CONFIG_S3C2440_NAND_BOOT) [Y/n/?] Y
    Support AND Boot (CONFIG_S3C2440_AND_BOOT) [N/y/?] y
*
* General setup
*
Define TEXT Address (CONFIG_VIVI_ADDR) [N/y/?] █
```

---

# 图形配置界面—make menuconfig



- 以图形界面配置 (make menuconfig) 为例，顶层 Makefile 调用 scripts/Menuconfig，按照 arch/S3C2440/config.in 来进行配置。命令执行完后产生文件 .config，其中保存着配置信息。下一次再做 make menuconfig 将产生新的 .config 文件，原 .config 被改名为 .config.old。

# arch/S3C2440/config.in解读

.....

```
mainmenu_option next_comment  
comment 'System Type'
```

```
choice 'ARM system type' \  
    "SA1100-based          CONFIG_ARCH_SA1100 \  
    PXA250/210-based      CONFIG_ARCH_PXA250 \  
    S3C2400-based         CONFIG_ARCH_S3C2400 \  
    S3C2410-based         CONFIG_ARCH_S3C2410 \  
    S3C2440-based         CONFIG_ARCH_S3C2440 "  
endmenu
```

# arch/S3C2440/config.in解读

```
mainmenu_option next_comment
comment 'Implementations'
if [ "$CONFIG_ARCH_S3C2400" = "y" ]; then
    .....
fi
if [ "$CONFIG_ARCH_S3C2410" = "y" ]; then
    choice 'Platform' \
        "SMDK      CONFIG_S3C2410_SMDK \
        MPORT3    CONFIG_S3C2410_MPORT3"
    if [ "$CONFIG_S3C2410_SMDK" = "y" ]; then
        bool '  Support NAND Boot' CONFIG_S3C2410_NAND_BOOT
        bool '  Support AMD Boot'  CONFIG_S3C2410_AMD_BOOT
    fi
fi
if [ "$CONFIG_ARCH_S3C2440" = "y" ]; then
    .....
fi
.....
```

# 菜单块定义

```
mainmenu_option next_comment  
comment 'System Type'  
...  
endmenu
```

# choice语句

choice 语句首先给出一串选择列表，供用户选择其中一种。格式为：

```
choice /prompt/ /word/ /word/
```

choice 首先显示 /prompt/，然后将 /word/ 分解成前后两个部分，前部分为对应选择的提示符，后部分是对应选择的配置变量。用户选择的配置变量为 y，其余的都为 n。因此，choice语句是一个多选一的结构



# 询问语句

## 1. 格式

`bool /prompt/ /symbol/`

`hex /prompt/ /symbol/ /word/`

`int /prompt/ /symbol/ /word/`

`string /prompt/ /symbol/ /word/`

`tristate /prompt/ /symbol/`

## 2. 说明

询问语句首先显示一串提示符 `/prompt/`，等待用户输入，并把输入的结果赋给 `/symbol/` 所代表的配置变量。不同的询问语句的区别在于它们接受的输入数据类型不同，比如 `bool` 接受布尔类型（`y` 或 `n`），`hex` 接受 16 进制数据。有些询问语句还有第三个参数 `/word/`，用来给出缺省值。

# source语句

## 1. 格式

`source /word/`

## 2. 说明

`/word/` 是文件名，`source` 的作用是调入新的文件。

因此，`lib/priv_data/Config.in`、`drivers/serial/Config.in`、`drivers/mtd/Config.in`和`lib/Config_cmd.in`文件会被调入，供用户进一步配置。

# PART SIX 典型嵌入式bootloader

## —vivi启动代码分析

# vivi代码分析

- 阶段1: `arch/s3c2410/head.S`
- 阶段2: `init/main.c`

# vivi代码分析—阶段1

- 1、关WATCH DOG：上电后，WATCH DOG默认是开着的
- 2、禁止所有中断：vivi中没用到中断(不过这段代码实在多余，上电后中断默认是关闭的)
- 3、初始化系统时钟：启动MPLL，FCLK=200MHz，HCLK=100MHz，PCLK=50MHz，“CPU bus mode”改为“Asynchronous bus mode”。
- 4、初始化内存控制寄存器
- 5、检查是否从掉电模式唤醒，若是，则调用WakeupStart函数进行处理——这是一段没用上的代码，vivi不可能进入掉电模式
- 6、点亮所有LED

# vivi代码分析—阶段1

- 7、初始化UART0

- a. 设置GPIO，选择UART0使用的引脚

- b. 初始化UART0，设置工作方式、波特率115200 8N1、无流控

8、将vivi所有代码(包括阶段1和阶段2)从nand flash复制到SDRAM

- a. 设置nand flash控制寄存器

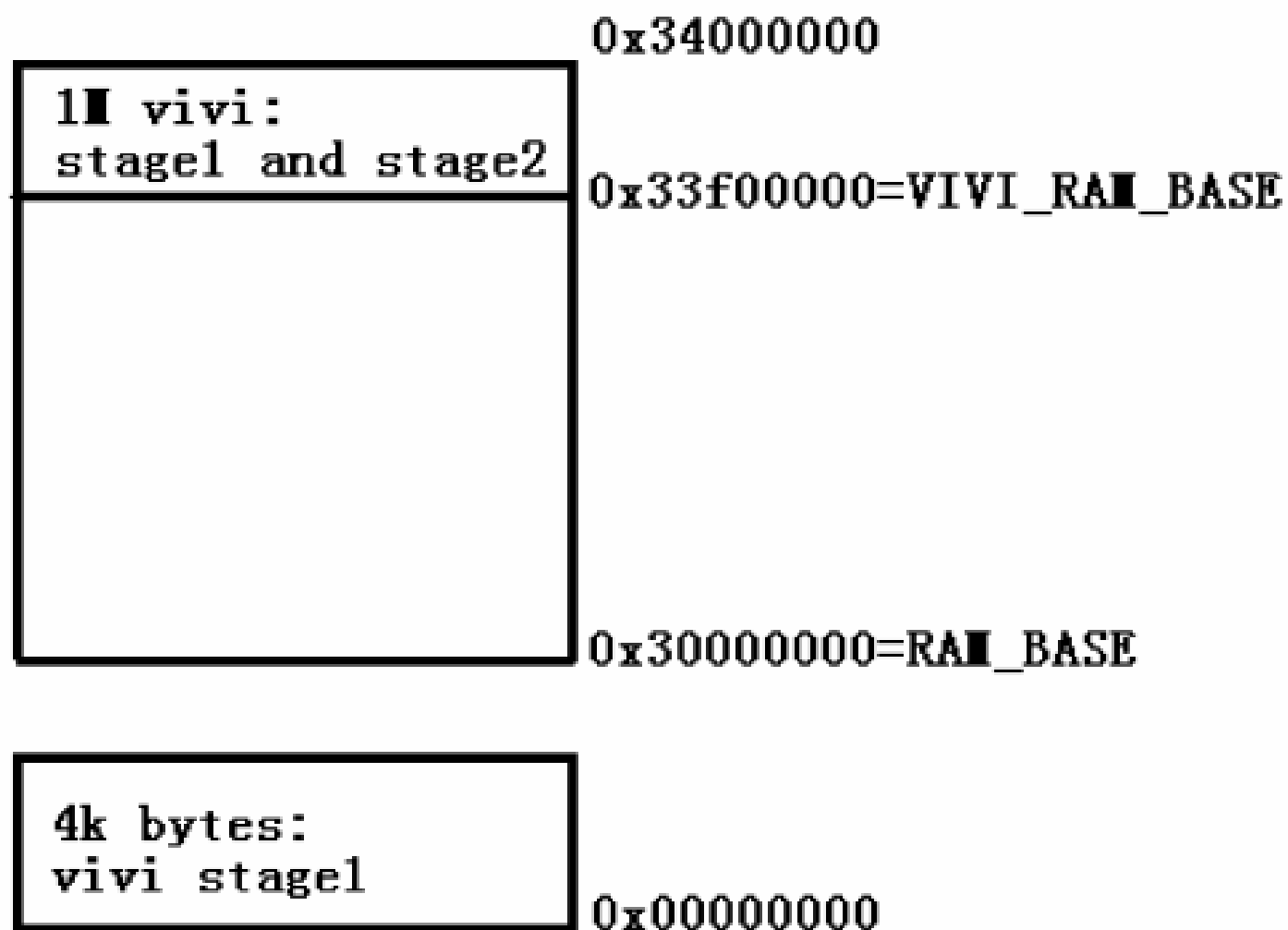
- b. 设置堆栈指针——调用C函数时必须先设置堆栈

- c. 设置即将调用的函数nand\_read\_ll的参数：r0=目的地址(SDRAM的地址)，r1=源地址(nand flash的地址)，r2=复制的长度(以字节为单位)

- d. 调用nand\_read\_ll进行复制

- e. 进行一些检查工作：上电后nand flash最开始的4K代码被自动复制到一个称为“Steppingstone”的内部RAM中(地址为0x00000000-0x00001000)；在执行nand\_read\_ll之后，这4K代码同样被复制到SDRAM中(地址为0x33f00000-0x33f01000)。比较这两处的4K代码，如果不同则表示出错

9、跳到bootloader的阶段2(main)运行



# vivi代码分析一阶段2 ( init/main.c )

- Step 1: reset\_handler()

reset\_handler用于将内存清零，代码  
lib/reset\_handle.c中

- Step 2: board\_init()

board\_init调用2个函数用于初始化定时器和设置各  
GPIO引脚功能，代码在arch/s3c2410/smdk.c

- Step 3: 建立页表和启动MMU

mem\_map\_init函数用于建立页表，vivi使用段式页  
表，只需要一级页表。它调用3个函数，代码在  
arch/s3c2410/mmu.c

- Step 4: heap\_init() heap

内存动态分配函数mmalloc就是从heap中划出一块空  
闲内存的，mfree则将动态分配的某块内存释放回  
heap中。



# vivi代码分析一阶段2 ( init/main.c )

- Step 5: `mtd_dev_init()`

Step 6: `init_priv_data()`

Step 7: `misc()`和`init_builtin_cmds()`

这两个函数都是简单地调用`add_command`函数，给一些命令增加相应的处理函数。

Step 8: `boot_or_vivi()`

此函数根据情况，或者启动“`vivi_shell`”，进入与用户进行交互的界面，或者直接启动linux内核。

## Step 1: reset\_handler()

```
reset_handler(void)
{
    int pressed;

    pressed = is_pressed_pw_btn();

    if (pressed == PWBT_PRESS_LEVEL) {
        DPRINTK("HARD RESET\r\n");
        hard_reset_handle();           /*清空内存*/
    } else {
        DPRINTK("SOFT RESET\r\n");
        soft_reset_handle();          /*空函数*/
    }
}
```

## Step 2: board\_init()

board\_init调用2个函数用于初始化定时器和设置各GPIO引脚功能，代码在arch/s3c2410/smdk.c中：

```
int board_init(void)
{

    init_time();
    set_gpios();

    return 0;
}
```

init\_time()只是简单的令寄存器TCFG0 = 0xf00，vivi未使用定时器，这个函数可以忽略。

set\_gpios()用于选择GPA-GPH端口各引脚的功能及是否使用各引脚的内部上拉电阻，并设置外部中断源寄存器EXTINT0-2

### Step 3: 建立页表和启动MMU

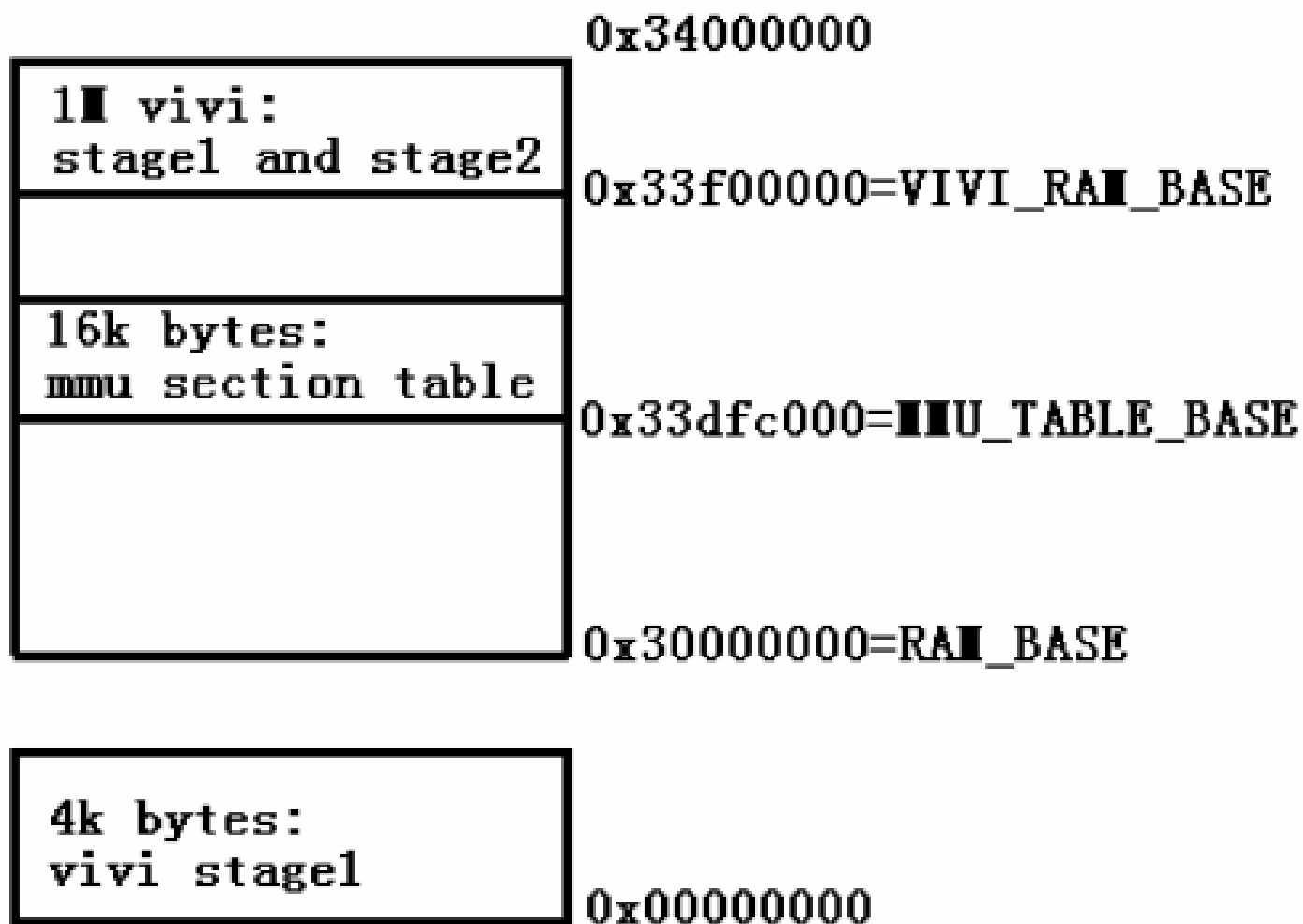
mem\_map\_init函数用于建立页表，vivi使用段式页表，只需要一级页表。它调用3个函数，代码在arch/s3c2410/mmu.c中

```
void mem_map_init(void)
{
#ifdef CONFIG_S3C2410_NAND_BOOT
    mem_map_nand_boot();
#else
    mem_map_nor();
#endif
    cache_clean_invalidate();
    tlb_invalidate();
}
```

mem\_map\_nand\_boot()函数调用mem\_mapping\_linear()函数来最终完成建立页表的工作。页表存放在SDRAM物理地址0x33dfc000开始处，共16K：一个页表项4字节，共有4096个页表项；每个页表项对应1M地址空间，共4G。

mem\_map\_init先将4G虚拟地址映射到相同的物理地址上，NCNB(不使用cache，不使用write buffer)——这样，对寄存器的操作跟未启动MMU时是一样的；再将SDRAM对应的64M空间的页表项修改为使用cache。

mmu\_init()函数用于启动MMU，它直接调用arm920\_setup()函数。  
arm920\_setup()的代码在arch/s3c2410/mmu.c中



## Step 4: heap\_init()

heap——堆，内存动态分配函数`malloc`就是从heap中划出一块空闲内存的，`free`则将动态分配的某块内存释放回heap中。

`heap_init`函数在SDRAM中指定了一块1M大小的内存作为heap(起始地址`HEAP_BASE = 0x33e00000`)，并在heap的开头定义了一个数据结构`blockhead`——事实上，heap就是使用一系列的`blockhead`数据结构来描述和操作的。每个`blockhead`数据结构对应着一块heap内存，假设一个`blockhead`数据结构的存放位置为A，则它对应的可分配内存地址为“A + `sizeof(blockhead)`”到“A + `sizeof(blockhead)` + size - 1”。

vivi对heap的操作比较简单，vivi中有一个全局变量`static blockhead *gHeapBase`，它是heap的链表头指针，通过它可以遍历所有`blockhead`数据结构。假设需要动态申请一块sizeA大小的内存，则`malloc`函数从`gHeapBase`开始搜索`blockhead`数据结构，如果发现某个`blockhead`满足：

- a. `allocated = 0` //表示未分配

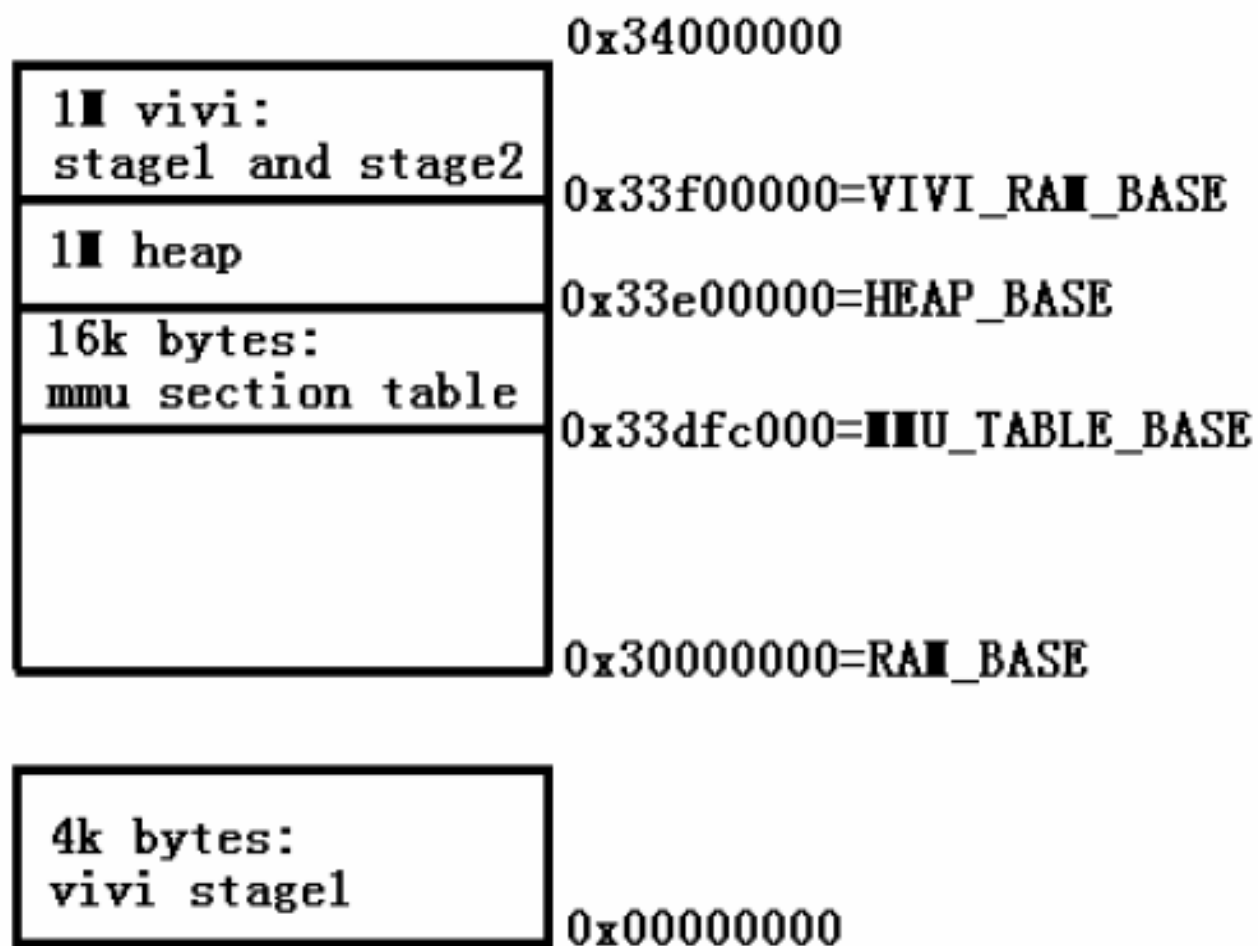
- b. `size > sizeA`,

则找到了合适的`blockhead`，于是进行如下操作：

- a. `allocated`设为1

- b. 如果`size - sizeA > sizeof(blockhead)`，则将剩下的内存组织成一个新的`blockhead`，放入链表中

- c. 返回分配的内存的首地址 释放内存的操作更简单，直接将要释放的内存对应的`blockhead`数据结构的`allocated`设为0即可。



## Step 5: mtd\_dev\_init()

在分析代码前先介绍一下MTD(Memory Technology Device)相关的技术。在linux系统中，我们通常会用到不同的存储设备，特别是FLASH设备。为了在使用新的存储设备时，我们能更简便地提供它的驱动程序，在上层应用和硬件驱动的中间，抽象出MTD设备层。驱动层不必关心存储的数据格式如何，比如是FAT32、EXT2还是FFS2或其它。它仅提供一些简单的接口，比如读写、擦除及查询。如何组织数据，则是上层应用的事情。MTD层将驱动层提供的函数封装起来，向上层提供统一的接口。这样，上层即可专注于文件系统的实现，而不必关心存储设备的具体操作。



## Step 6: `init_priv_data()`

此函数将启动内核的命令参数取出，存放在内存特定的位置中。这些参数来源有两个：vivi预设的默认参数，用户设置的参数（存放在nand flash上）。`init_priv_data`先读出默认参数，存放在“VIVI\_PRIV\_RAM\_BASE”开始的内存上；然后读取用户参数，若成功则用用户参数覆盖默认参数，否则使用默认参数。

`init_priv_data`函数分别调用`get_default_priv_data`函数和`load_saved_priv_data`函数来读取默认参数和用户参数。这些参数分为3类：

- a. vivi自身使用的一些参数，比如传输文件时的使用的协议等
- b. linux启动命令
- c. nand flash的分区参数

`get_default_priv_data`函数比较简单，它将vivi中存储这些默认参数的变量，复制到指定内存中。

1M vivi	0x3400_0000 = 64M
1M heap	0x33f0_0000 = VIVI_RAM_BASE
16K MMU	0x33e0_0000 = HEAP_BASE
16K linux启动命令 16K PARAMETER_TLB 16K 分区表参数	0x33df_c000 = MMU_TABLE_BASE LINUX_CMD_OFFSET = 16k PARAMETER_TLB_OFFSET = 16k
32K stack	0x33df_0000 = VIVI_PRIV_RAM_BASE (PARAMETER_PART_OFFSET = 0)
	0x33de_8000 = STACK_BASE

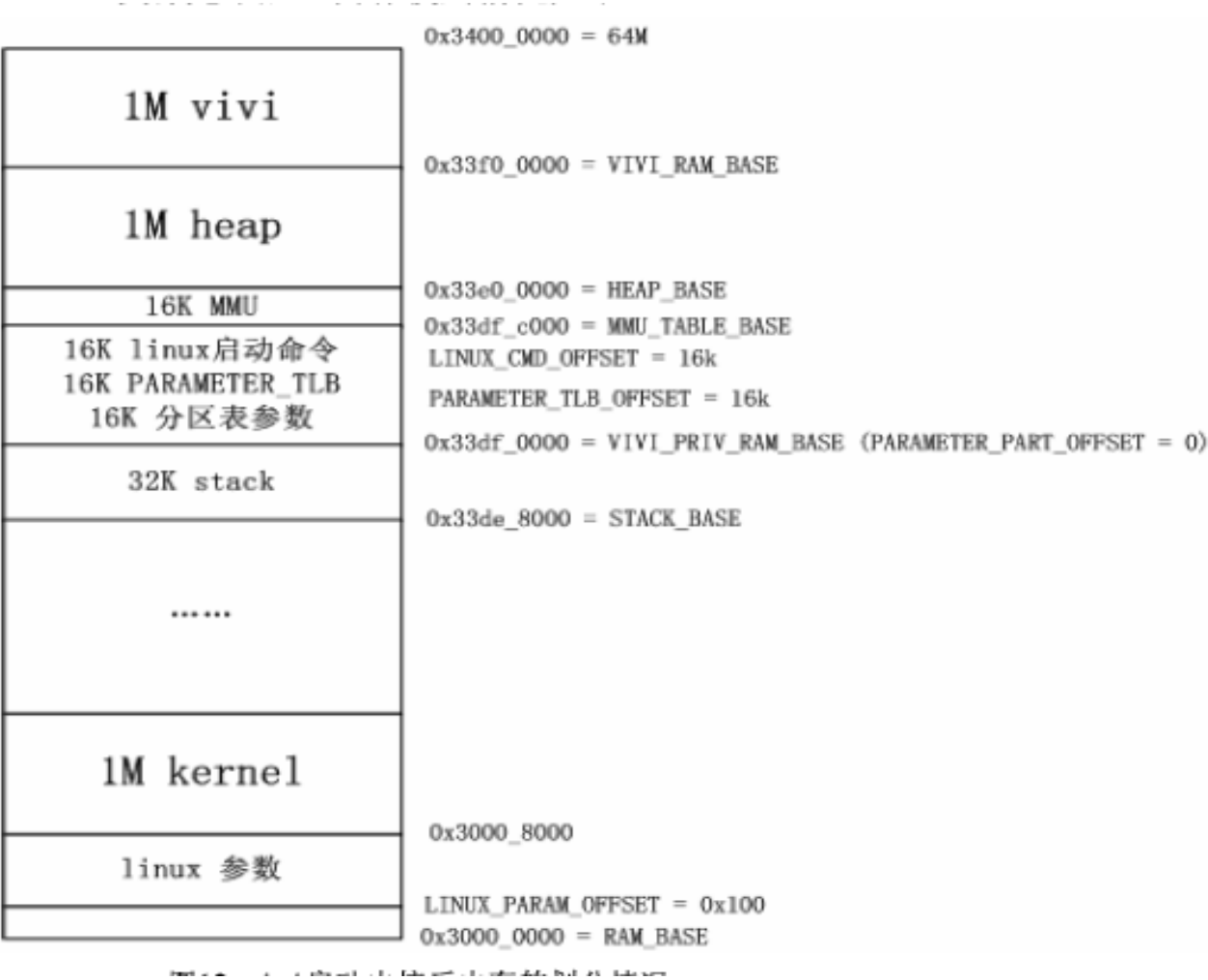
## step 7: misc()和init\_builtin\_cmds()

这两个函数都是简单地调用add\_command函数，给一些命令增加相应的处理函数。在vivi启动后，可以进去操作界面，这些命令，就是供用户使用的。它们增加了如下命令：

- a. add\_command(&cpu\_cmd)
- b. add\_command(&bon\_cmd)
- c. add\_command(&reset\_cmd)
- d. add\_command(&param\_cmd)
- e. add\_command(&part\_cmd)
- f. add\_command(&mem\_cmd)
- g. add\_command(&load\_cmd)
- h. add\_command(&go\_cmd)
- i. add\_command(&dump\_cmd)
- j. add\_command(&call\_cmd)
- k. add\_command(&boot\_cmd)
- l. add\_command(&help\_cmd)

Step 8: boot\_or\_vivi()

此函数根据情况，或者启动“vivi\_shell”，进入与用户进行交互的界面，或者直接启动linux内核。



PART SEVEN

典型嵌入式bootloader

—实验

# 实验1: vivi裁减编译烧写

- make menuconfig

```
SECTIONS {  
  . = 0x33f00000;  
  .text          : { *(.text) }  
  .data ALIGN(4) : { *(.data) }  
  .bss ALIGN(4)  : { *(.bss)  *(COMMON) } }
```

#####

vivi的使用熟悉

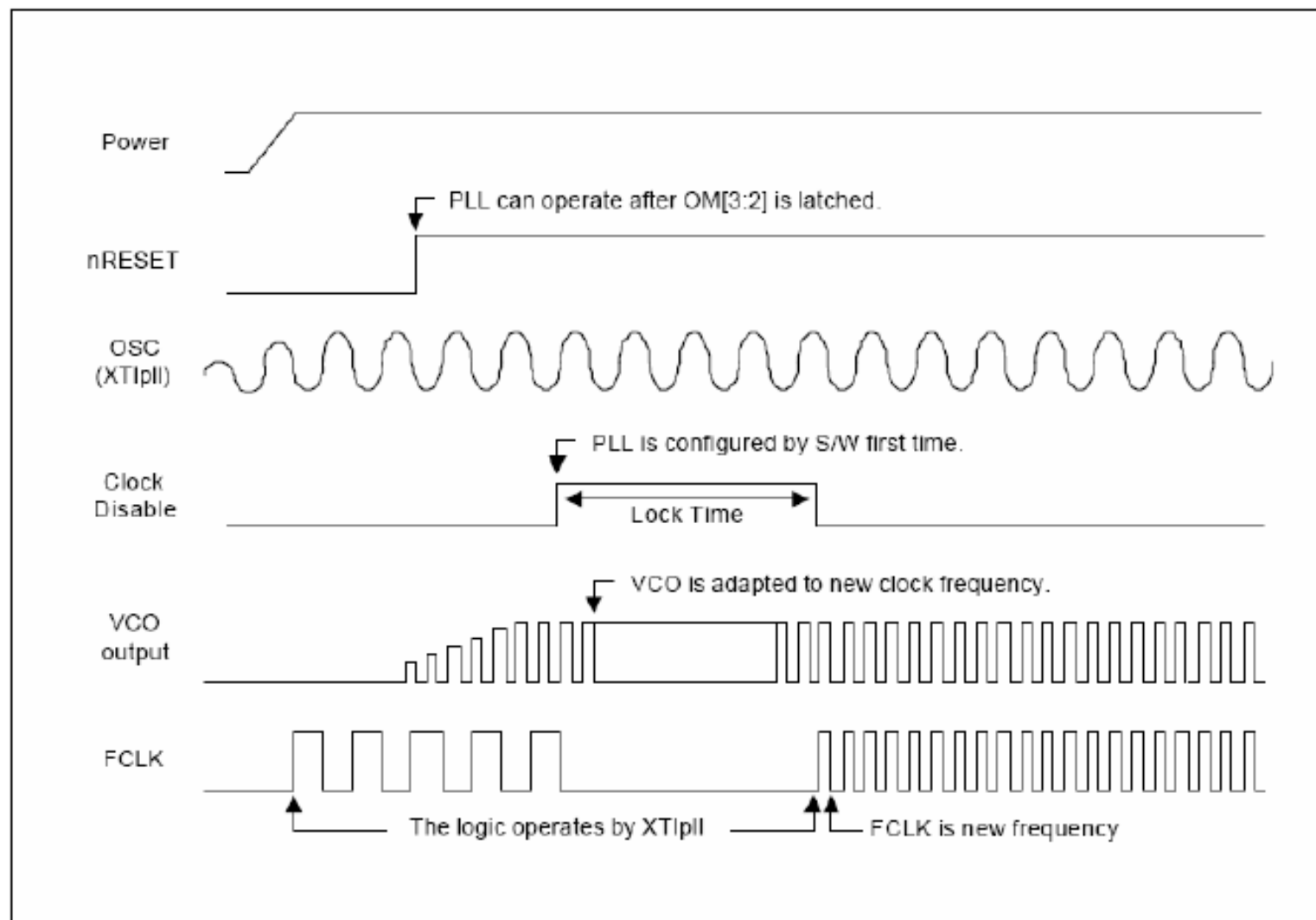
烧写vivi, kernel (zImage)

修改vivi的参数

## 实验2: clock调整实验

- S3C2410 CPU主频可以达到266MHz, 前面的实验都没有使用PLL, CPU的频率只有12MHz。本实验启动PLL, 使得FCLK=200MHz, HCLK=100MHz, PCLK=50MHz。

S3C2410有两个PLL: MPLL和UPLL, UPLL专用与USB设备, 本实验介绍的是MPLL——用于设置FCLK、HCLK、PCLK。FCLK用于CPU核, HCLK用于AHB总线的设备(比如SDRAM), PCLK用于APB总线的设备(比如UART)。





设置MPLL的几个寄存器:

1、LOCKTIME: 设为0x00ffffff

前面说过, MPLL启动后需要等待一段时间(Lock Time), 使得其输出稳定。位[23:12]用于UPLL, 位[11:0]用于MPLL。

本实验使用确省值0x00ffffff。

2、CLKDIVN: 设为0x03

用于设置FCLK、HCLK、PCLK三者的比例:

bit[2]——HDIVN1, 若为1, 则bit[1:0]必须设为0b00, 此时FCLK:HCLK:PCLK=1:1/4:1/4; 若为0, 三者比例由bit[1:0]确定

bit[1]——HDIVN, 0: HCLK=FCLK; 1: HCLK=FCLK/2

bit[0]——PDIVN, 0: PCLK=HCLK; 1: PCLK=HCLK/2

本实验设为0x03, 则FCLK:HCLK:PCLK=1:1/2:1/4

3、请翻到数据手册226页，有这么一段： If HDIVN = 1, the CPU bus mode has to be changed from the fast bus mode to the asynchronous bus mode using following instructions:

MMU\_SetAsyncBusMode

```
mrc p15, 0, r0, c1, c0, 0
orr r0, r0, #R1_nF:0R:R1_iA
mcr p15, 0, r0, c1, c0, 0
```

其中的“R1\_nF:0R:R1\_iA”等于0xc0000000。意思就是说，当HDIVN = 1时，CPU bus mode需要从原来的“fast bus mode”改为“asynchronous bus mode”。

4、MPLLCON: 设为  $(0x5c \ll 12) \mid (0x04 \ll 4) \mid (0x00)$ ，即 0x5c0040 对于MPLLCON寄存器，[19:12]为MDIV，[9:4]为PDIV，[1:0]为SDIV。有如下计算公式： $MPLL(FCLK) = (m * Fin) / (p * 2^s)$  其中： $m = MDIV + 8$ ， $p = PDIV + 2$  对于本开发板， $Fin = 12MHz$ ，MPLLCON设为0x5c0040，可以计算出FCLK=200MHz，再由CLKDIVN的设置可知：HCLK=100MHz，PCLK=50MHz。

当工作频率改变后，一些设备的初始化参数需要调整，很幸运，只需要修改两个地方：

1、SDRAM控制器的REFRESH寄存器(在init.c中)：

MEMORY CONTROLLER中关于REFRESH寄存器的介绍，本实验HCLK=100MHz，REFRESH寄存器取值如下：

$$R\_CNT = 2^{11} + 1 - 100 * 7.8125 = 1268,$$

$$REFRESH = 0x008e0000 + 1268 = 0x008e04f4$$

在init.c中的memsetup\_2函数，将原来的0x008e07a3换成0x008e04f4即可。

2、UART0的UBRDIV0寄存器(在serial.c中)：

UART中关于UBRDIV0的介绍，本实验PCLK=50MHz，设置波特率为57600时，UART0寄存器取值可由下式计算得53：

$$UBRDIVn = (\text{int})(PCLK / (\text{bps} \times 16)) - 1$$

在serial.c中的init\_uart函数，将UART0由原来的12换成53即可。

# LAB3 添加命令到vivi中

- 添加命令
  - ledon : 点亮led上的四盏灯
  - ledoff : 熄灭led上的四盏灯

PART EIGHT

典型嵌入式bootloader

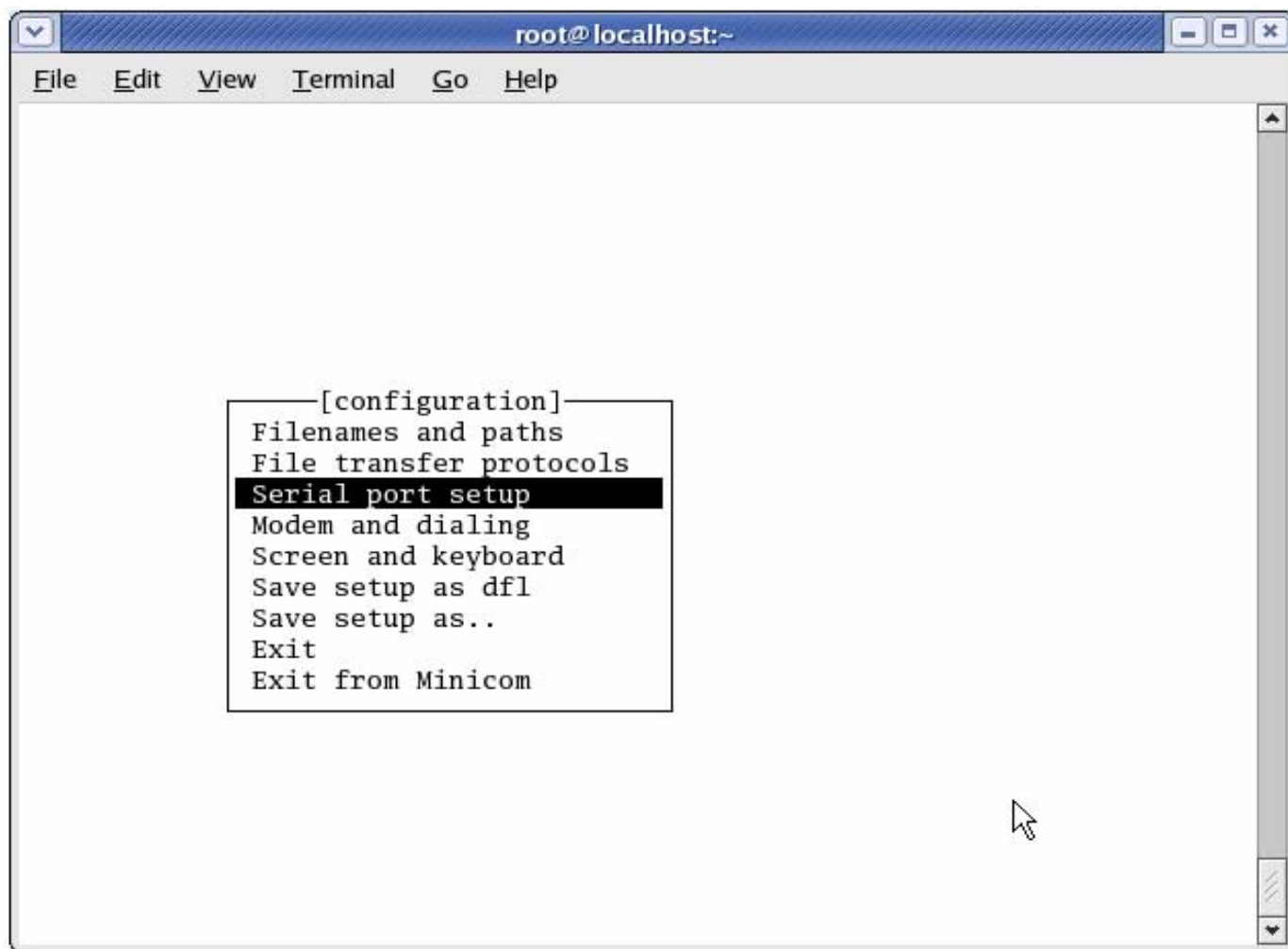
—UBOOT使用

# 概述

- Minicom配置
- 启动TFTP服务器
- U-BOOT.BIN固化到ARMSYS2410
- 配置U-BOOT
- 通过TFTP传输并写入映像文件

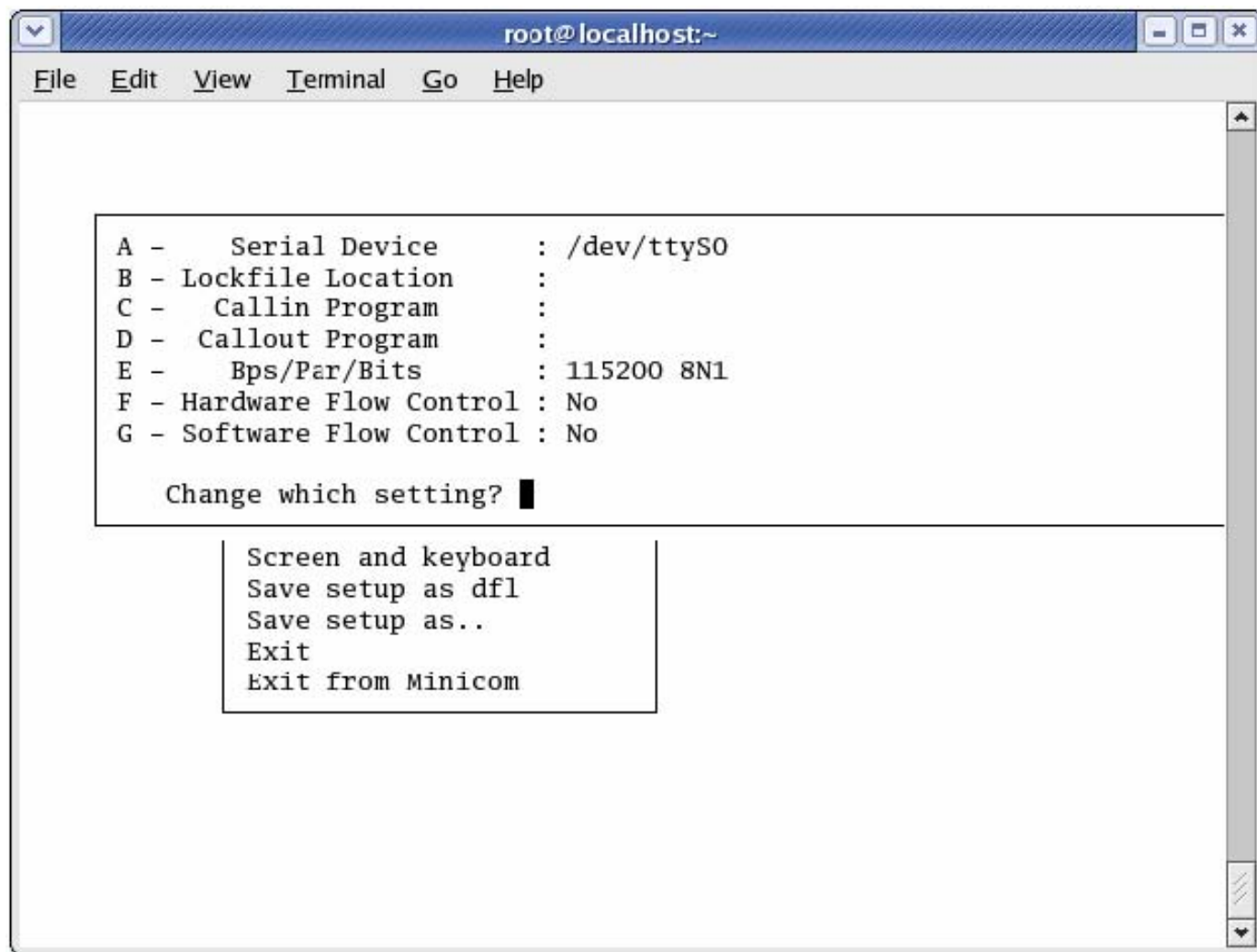
# 1.配置minicom

- [root@localhost root]# minicom -s
- 在设置模式下执行minicom



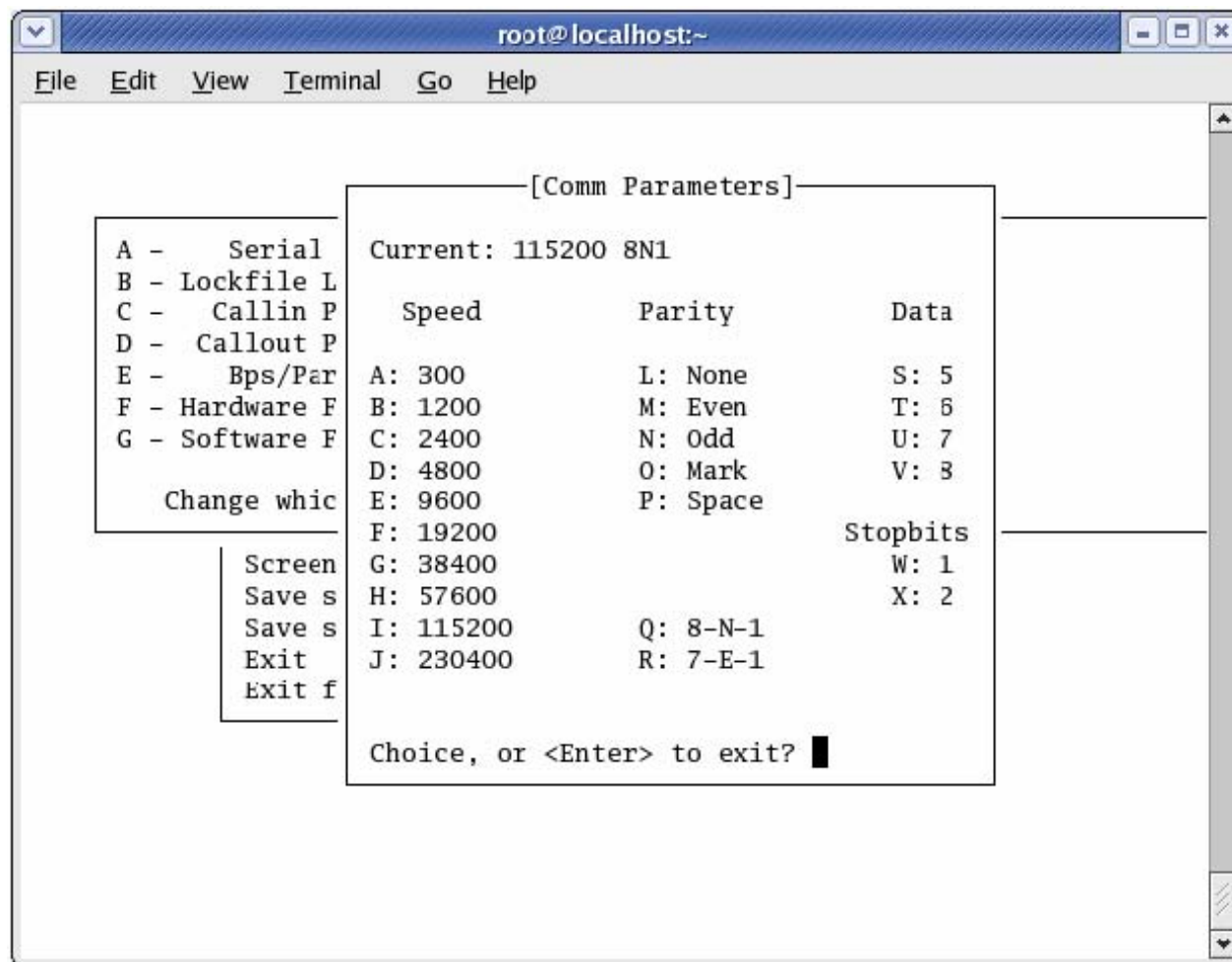
请选择 ‘串口配置图’，然后按下 ‘A’ 键开始设置

‘**Serial Device**’，输入与开发板连接的串口名称。（如果采用 **COM1**，输入 **/dev/ttyS0**，如果采用 **COM2**，输入 **/dev/ttyS1**。）

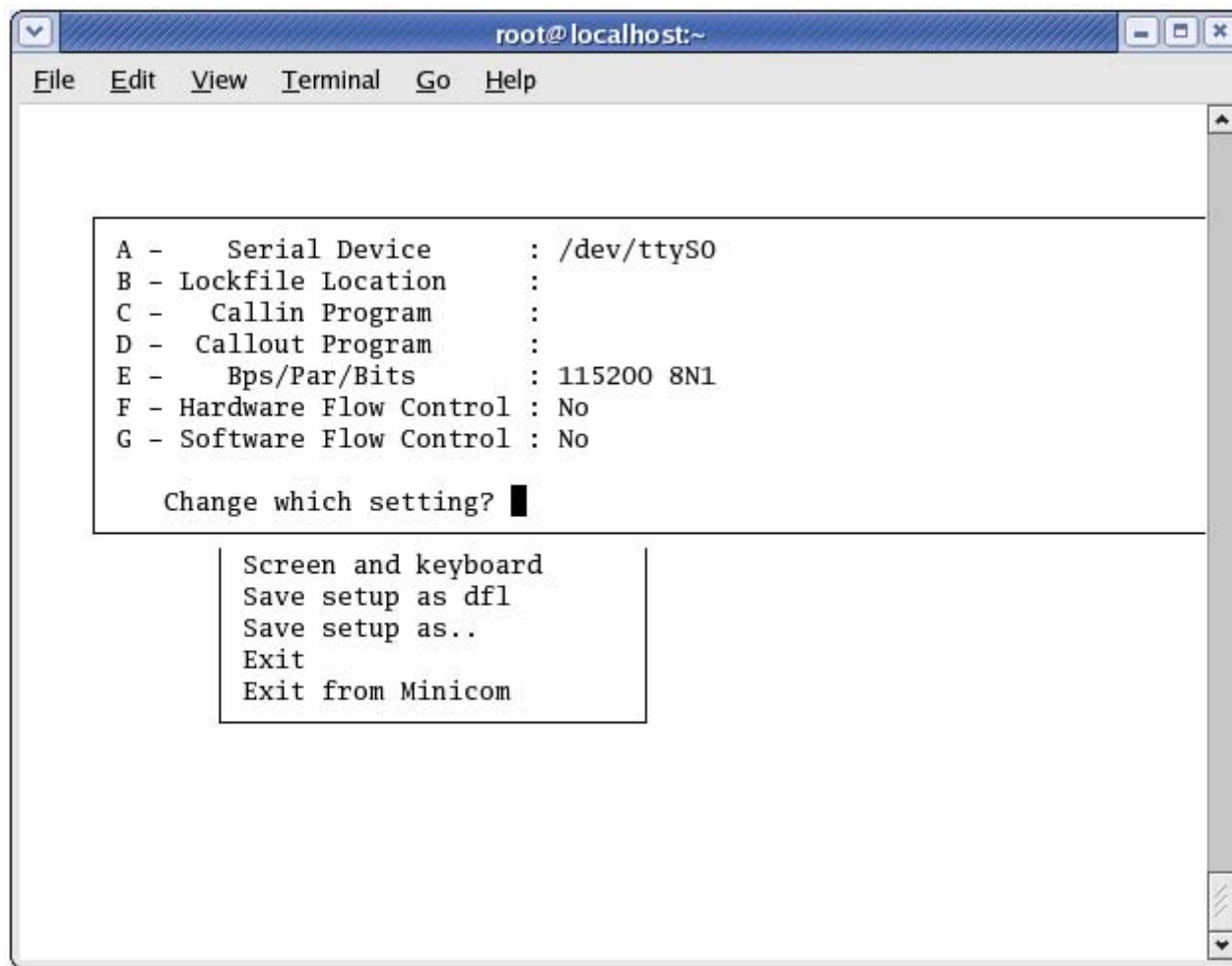


按下 ‘E’ 键来设置  
‘**bps/Par/Bits**’.  
按下 ‘I’ 来设置  
‘**bps**’ 为  
**115200**, 按下 ‘V’  
来设置 ‘**Data**  
**bits**’ 为 **8**, 按下  
‘W’ 来设置 ‘**S**为**p**  
**bits**’ 为 ‘**1**’, 按下  
‘V’ 来设置  
‘**parity**’ 为  
‘**NONE**’.

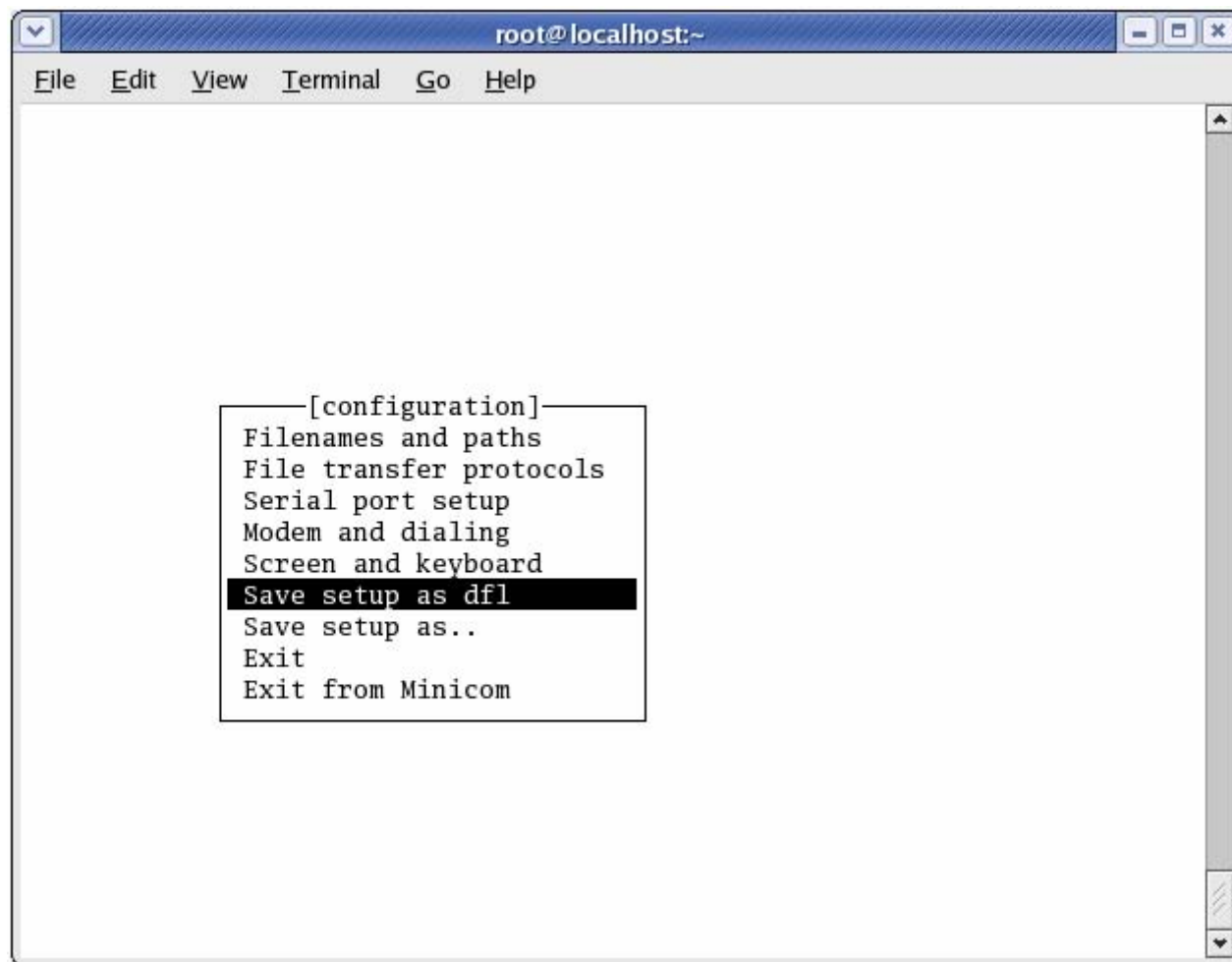




按下 ‘E’ 键来设置  
‘bps/Par/Bits’.  
按下 ‘I’ 来设置  
‘bps’ 为  
115200, 按下 ‘V’  
来设置 ‘Data  
bits’ 为 8, 按下  
‘W’ 来设置 ‘S为p  
bits’ 为 ‘1’, 按下  
‘V’ 来设置  
‘parity’ 为  
‘NONE’.

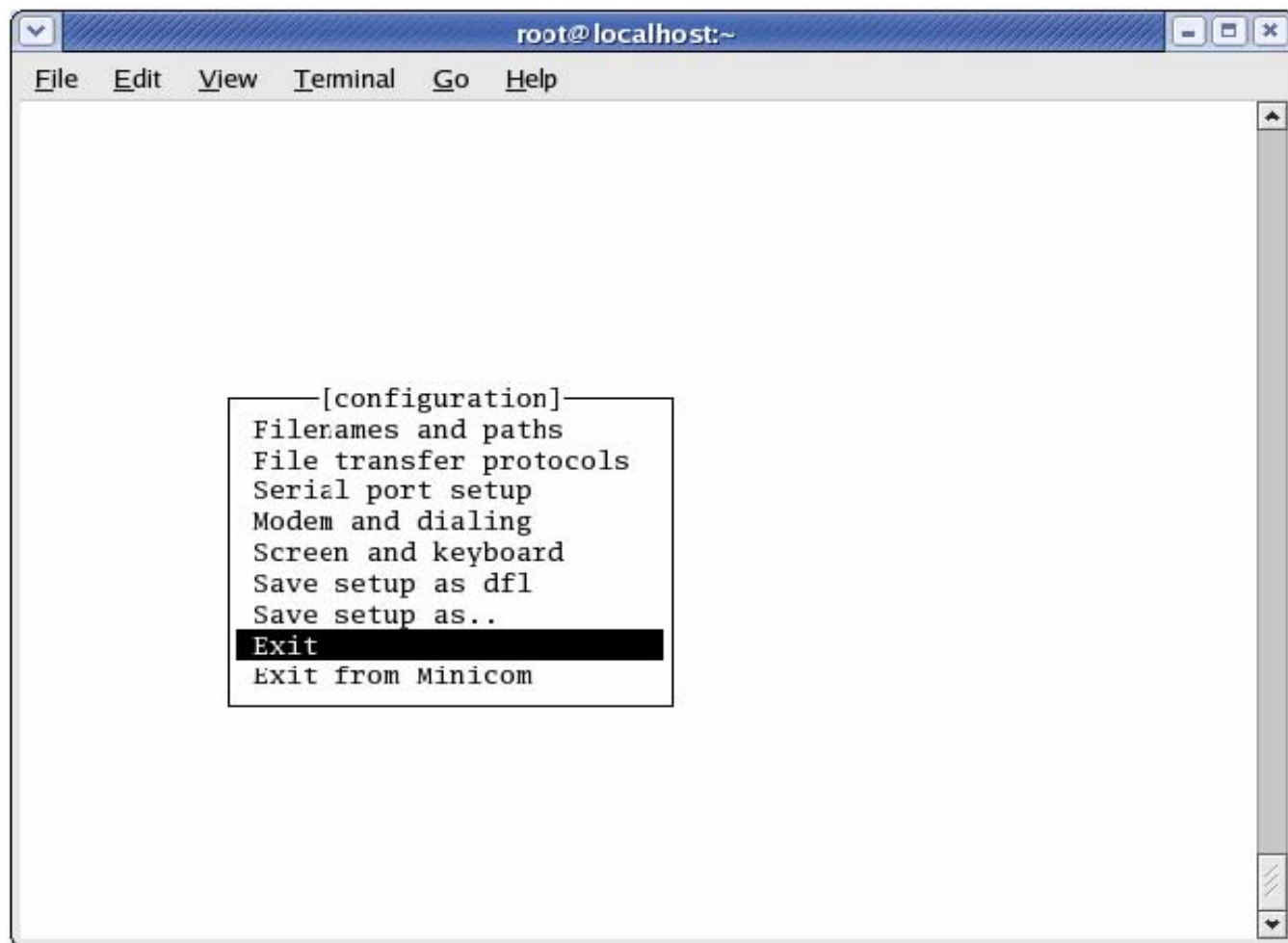


按下‘F’ 键设置  
**‘Hardware Flow Control’** 为 **‘NO’**.  
按下‘G’ 键设置  
**‘Software Flow Control’** 为 **‘NO’**.  
缺省值也是 **‘NO’**.



配置结束，按下‘回车’键。选择‘**Save setup as dfl**’项目，然后按下‘回车’来保存刚刚设置的值。

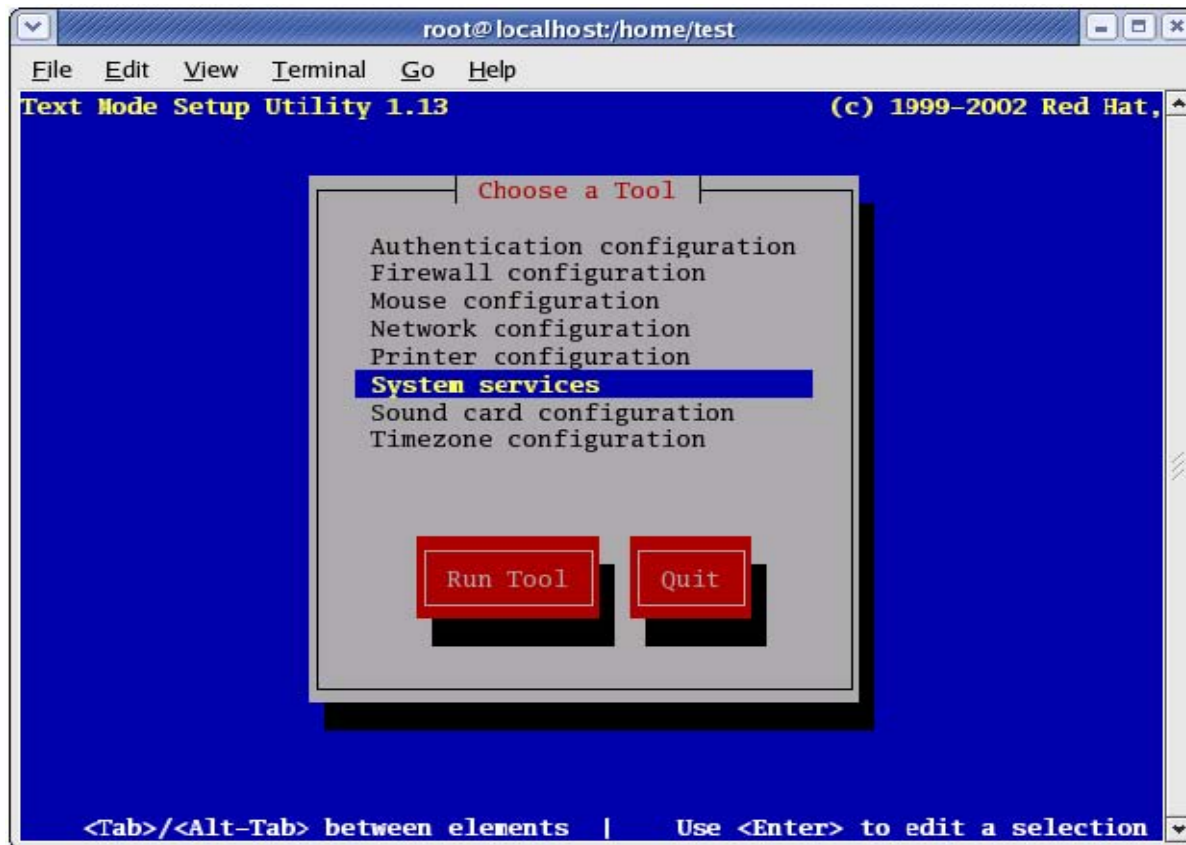
按下‘Exit’键，从配置状态退出。现在配置被保存到文件‘/etc/minirc.dfl’中。



按下‘Exit’ 键，  
从配置状态退出。  
现在配置被保存到文件  
‘/etc/minirc.dfl’  
中。

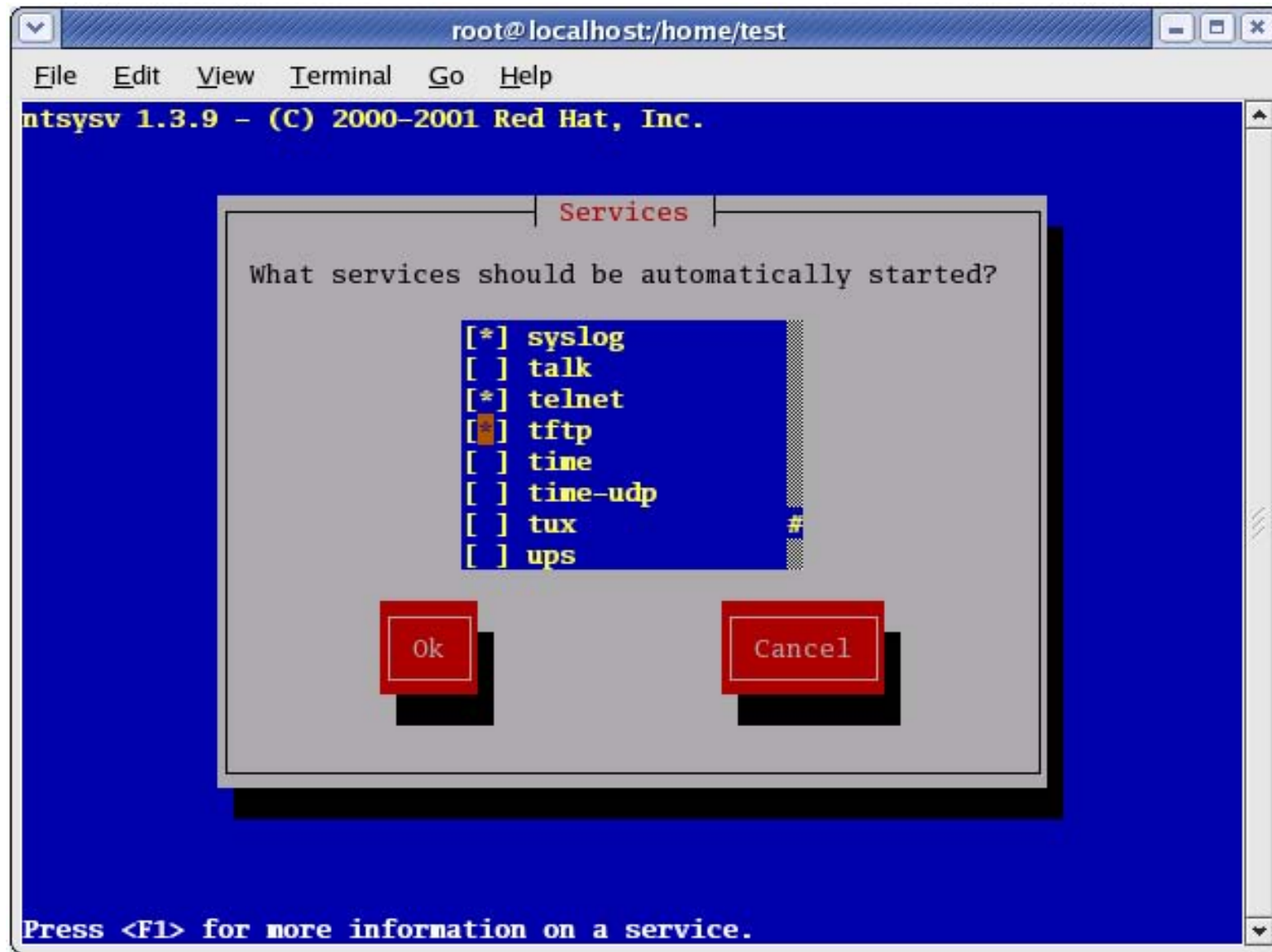
## 2.配置tftp服务器

- 为了采用使用 tftp 服务器程序，必须通过以下命令来设置主机：
- **[root@localhost root]# setup**
- 可以看到如下图所示的 “Text Mode Setup Utility”：



请选中

“System services”



请选中 “**tftp**” 服务，然后按下 “**ok**” 退出。

最后按下 “**quit**” 退出配置工具。

执行以下命令行：

```
[root@localhost root]# xinetd -restart
```

现在你可以采用**u-boot**，通过**tftp**服务下载编译好的映像文件。在下载之前，用以太网线连接**PC**主机和**ARMSYS2410**开发板。

### 3. u-boot.bin固化到开发板

- 用ARMSYS2410套件中提供的并口线连接JTAG小板，再用20针排线连接JTAG小板和ARMSYS2410开发板。
- 令开发板上电。将u-boot.bin文件与Jflash-s3c2410放在同一目录下。
- 开始烧录：
  - **cd Jflash**
  - **./Jflash-s3c2410 u-boot.bin /t=5**



终端上出现：

```
+-----+ | SEC JTAG FLASH(SJF) v 0.11
+ | modified by MIZI 2002.7.13 + +-----
-----+ > flashType=5 > S3C2410X(ID=0x0032409d) is
detected. > K9S1208 is detected. ID=0xec76
```

K9S1208 NAND Flash JTAG Programmer Ver 0.0 0:K9S1208  
Program 1:K9S1208 Pr BlkPage 2: Exit

Select the function to test : **0** :这里输入“0”.

[SMC(K9S1208) NAND Flash Writing Program]

Source size: 0xe4af

Available target block number: 0~4095 Input target block number: **0** : 这里输入 "0".

target start block number =0 target size (0x4000\*n) =0x10000

[illegible][illegible]

E p

E p

K9S1208 NAND Flash JTAG Programmer Ver 0.0 0:K9S1208  
Program 1:K9S1208 Pr BlkPage 2: Exit

Select the function to test : **2** : 输入“2”退出烧录.

- 这样，u-boot.bin就被成功地烧录到了flash中。关闭开发板电源，拔除JTAG小板，然后用以太网线、串口线连接PC机和开发板。重新复位开发板，可以从minicom上看到如下输出信息：



```
root@localhost:~  
文件(F)  编辑(E)  查看(V)  终端(T)  转到(G)  帮助(H)  
  
U-Boot 1.0.0 (Feb 21 2006 - 17:42:41)  
  
U-Boot code: 33F00000 -> 33F1971C  BSS: -> 33F2D054  
IRQ Stack: 33f4e050  
FIQ Stack: 33f4f050  
  
DRAM Configuration:  Bank #0: 30000000 64 MB  
NAND: 64 MB  
  
-----  
OEM name       :  LiYuTai Elec.Co.,Ltd.  
Website        :  www.hzlitai.com.cn  
Email          :  lyt_tech@yahoo.com.cn  
Function       :  ARMSYS's BIOS for S3C2410A  
UART config    :  115.2kbps,8Bit,NP,UART0  
-----  
  
*** Warning - bad CRC, using default environment  
  
In:    serial  
Out:   serial  
Err:   serial  
SMDK2410 #
```

## 4.配置UBOOT

- 配置**PC**主机的**IP**
  - **[root@localhost tftpboot]# ifconfig eth0 down**
  - **[root@localhost tftpboot]# ifconfig eth0  
192.168.253.1 netmask 255.255.255.0 up**
  - **[root@localhost tftpboot]# ifconfig**
- 将主机的**IP**设置为**192.168.253.1**。也可以设置为任何你需要的**IP**，但要保证与开发板处于同一网段。

- 配置**ARMSYS2410**的**u-boot**包括以下项目，第一次使用时必须正确配置否则无法正常工作：
  - 开发板自身**IP**，
  - **Tftp**服务器**IP**，
- — 内核启动参数
  - **Bootcmd**启动命令
- 可以采用命令 “**setenv**”来设置参数, “**saveenv**”来保存参数
  - **SMDK2410 # setenv ipaddr 192.168.253.8**
  - **SMDK2410 # setenv serverip 192.168.253.1**
  - **SMDK2410 # setenv linux\_arg noinitrd root=/dev/mtdblock/2  
init=/linuxrc console=ttyS0**
  - **SMDK2410 # setenv bootcmd nandr c e0000 30008000\; bootm**
  - **SMDK2410 # saveenv**

# 5.通过**tftp**传输并写入映像文件

- 下载并更新**u-boot.bin**
  - **SMDK2410 # tftp <temporary address> <image name>**
  - **SMDK2410 # tftp 30000000 u-boot.bin**
  - 成功运行以上命令要求u-boot.bin文件已经被放在主机的/tftpboot目录下。
  - 然后将下载的 ***u-boot.bin***写入到Nandflash。运行以下命令：
  - **SMDK2410 # nandw <start block number> <image size> <temporary address>**
  - **SMDK2410 # nandw 0 20000 30000000**
  - 上面的300000000是SDRAM的暂存地址：0x30000000。
  - u-boot.bin位于Nandflash的起始块号为 0，也就是NANDflash的第一个块。
  - u-boot的大小一般小于128K（0x20000），因此最多占用8个块（0~7, 0x20000）。

# 下载并更新下载并更新**zImage**

- **SMDK2410** # tftp 300000000 zImage
- 通过以下命令将zImage写入Nandflash:
- **SMDK2410** # nandw c e0000 300000000
- 这里使用的暂存地址是0x300000000。
- 内核的起始块号为c。8~b块是存放环境变量的空间。

PART NINE

典型嵌入式bootloader

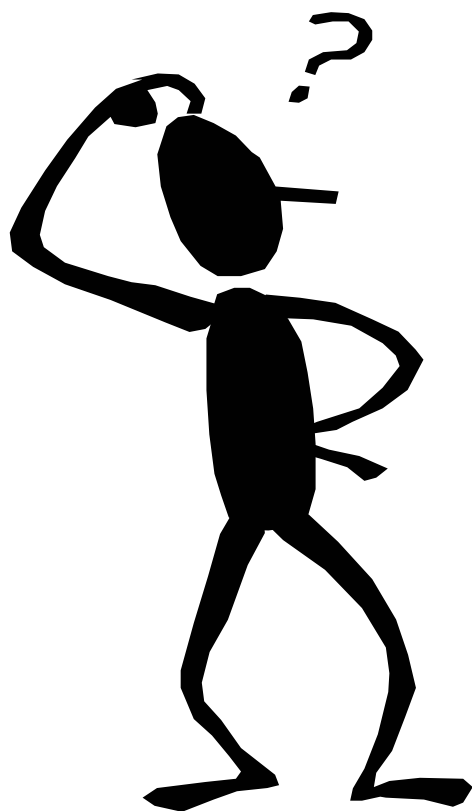
—UBOOT实验

# UBOOT实验一

- 要求： UBOOT加载实验
- 配置环境
- 将UBOOT加载到SDRAM,并烧写到  
NANDFLASH



让我们一起讨论！



谢谢！