网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细,供大家参考。

也许有错漏,请自行分辨。

目录

根文件系统启动分析

启动根文件系统过程

UBOOT:启动内核
内核：启动应用程序
应用程序又是根文件系统上。（要挂接根文件系统）

```
start_kernel
    setup_arch          // 解析u-boot传入的启动参数
    setup_command_line  // 解析u-boot传入的启动参数
    parse_early_param
        do_early_param
            从__setup_start到__setup_end，调用early函数
    unknown_bootoption
        obsolete_checksetup
            从__setup_start到__setup_end，调用非early函数
    rest_init
       kernel_init
          prepare_namespace
              mount_root     // 挂接根文件系统
          init_post
              // 执行应用程序
```

执行应用程序在“init_post”这个函数中。

```
00748: static int noinline init_post(void)
00749: {
00750:     free_initmem();
00751:     unlock_kernel();
00752:     mark_rodata_ro();
00753:     system_state = SYSTEM_RUNNING;
00754:     numa_default_policy();
00755:
00756:     if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
00757:         printk(KERN_WARNING "Warning: unable to open an initial console.\n");
00758:
00759:     (void) sys_dup(0);
00760:     (void) sys_dup(0);
00761:
00762:     if (ramdisk_execute_command) {
00763:         run_init_process(ramdisk_execute_command);
00764:         printk(KERN_WARNING "Failed to execute %s\n",
00765:                 ramdisk_execute_command);
00766:     }
00767:
00768:     /*
00769:      * We try each of these until one succeeds.
00770:      * The Bourne shell can be used instead of init if we are
00771:      * trying to recover a really broken machine.
00772:      */
00773:     if (execute_command) {
00774:         run_init_process(execute_command);
00775:         printk(KERN_WARNING "Failed to execute %s.  Attempting "
00776:                     "defaults...\n", execute_command);
00777:     }
00778:     run_init_process("/sbin/init");
00779:     run_init_process("/etc/init");
00780:     run_init_process("/bin/init");
00781:     run_init_process("/bin/sh");
00782:
00783:     panic("No init found.  Try passing init= option to kernel.");
00784: } ? end init_post ?
```

```
sys_open((const char __user *) "/dev/console", O_RDWR, 0
```

这里打开 /dev/console 设备。

内核如何打开第一个应用程序：

1. **打开一个设备：**

```
open(dev/console).
    (void) sys_dup(0);
                (void) sys_dup(0);
```

(这三个文件代表标准输入，标准输出，标准错误)

写程序时经常用到 printf 标准输出,scanf 标准输入,err()标准错误。它们是指 3 个文件。

假设 open(dev/console) 是打开的第 0 个文件。sys_dup(0)是指复制的意思，

复制后是第 1 个文件，则有了第 0 个文件和第 1 个文件。第 1 个文件也是指向 dev/console

另一个 sys_dup(0) 也是复制文件，则这个是第 3 个文件，也是指向 dev/console 这个文件。

意思是，所有的 printf 打印信息都会从 dev/console 打印出来。
输入时也是从 dev/console 这个文件输入的。想输入错误信息也是从 dev/console 这个文件输出的。这个文件 dev/console 对应终端。在我们这个例子中 dev/console 对应串口 0.
对于其他设备，这个 dev/console 可能是 键盘液晶。

2.用 run_init_process 启动第一个应用程序：

一般来说第一个应用程序运行后，就不会从系统返回了。
这个应用程序：

（1）要么是 UBOOT 传进来的命令行参数： init=linuxrc 这个样子

（2）要么是 sbin/init,要是 sbin/init 不成功，则还有 /etc/init 或 bin/init,bin/sh

```
00773:      if (execute_command) {
00774:          run_init_process(execute_command);
00775:          printk(KERN_WARNING "Failed to execute %s.  Attempting "
00776:                      "defaults...\n", execute_command);
00777:      }
```

如果有 execute_command 则执行：run_init_process(execute_command);
查找 execute_command 可知：

```
static int __init init_setup(char *str)
{
    unsigned int i;

    execute_command = str;
    /*
     * In case LILO is going to boot us with default command line,
     * it prepends "auto" before the whole cmdline which makes
     * the shell think it should execute a script with such name.
     * So we ignore all arguments entered _before_ init=... [MJ]
     */
    for (i = 1; i < MAX_INIT_ARGS; i++)
        argv_init[i] = NULL;
    return 1;
}
__setup("init=", init_setup);
```

__setup("init=", init_setup);这是 UBOOT 传递给内核的命令行参数。如果涉及了 "init=" 什么东西，显然这个 execute_command 就是等于 "init=" 后面的东西。

```
bootargs=noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
```

如果设置了 init= 某个文件（linuxrc） ,那么这个 execute_command 就等于这个文件(linuxrc).

```
if (execute_command) {
run_init_process(execute_command);
printk(KERN_WARNING "Failed to execute %s.  Attempting "
"defaults...\n", execute_command);
}
```

则上面的意思是：如果定义了 execute_command ，则会用 run_init_process 启动这个应用

程序 execute_command。
run_init_process(execute_command);若这个程序没有死掉没有返回，那么函数 run_init_process 就不会返回。我们第一个程序一般来说都是一个死循环。
若没有定义 execute_command，则往下执行：

```
00778:      run_init_process("/sbin/init");
00779:      run_init_process("/etc/init");
00780:      run_init_process("/bin/init");
00781:      run_init_process("/bin/sh");
```

第一个程序一般是死循环，一旦执行了 /sbin/init 这样的程序，就一去不复返了。如这里若是执行的第一个程序是 "/sbin/init",则就不返回了，不会执行下面的 /etc/init 等另几个可能作为第一个应用程序的程序。（这里是/sbin/init 执行不成功，便执行/etc/init。这个也不成功，则继续去执行/bin/init 等）。

测试下：

先在 UBOOT 中擦除文件系统。nand erase root

再启动 boot

```
VFS: Mounted root (yaffs filesystem).
Freeing init memory: 136K
Warning: unable to open an initial console.
Failed to execute /linuxrc.  Attempting defaults...
Kernel panic - not syncing: No init found.  Try passing init= option to kernel.
```

这是删除掉根文件系统时的打印信息。
要打开应用程序，首先要挂接根文件系统。打印信息中显示已经挂接上根文件系统了：

```
VFS:Mounted root(yaffs filesystem)
```

对于 FLASH 来说，把 FLASH 擦除了就相当于格式化了。你可以把它当成任何一种文件系统。这里显示是默认当成 yaffs filesystem。这里已经挂接上去了。但是里面是没有内容的。就没法启动应用程序了。这样打开显然是会失败的：

```
if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
    printk(KERN_WARNING "Warning: unable to open an initial console.\n");
```

（见上面错误信息）

再看：

```
bootargs=noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
```

可见 execute_command 定义的是 init=/linuxrc ，擦除了 FLASH 后也找不到了。故错误：

```
if (execute_command) {
    run_init_process(execute_command);
    printk(KERN_WARNING "Failed to execute %s.  Attempting "
                "defaults...\n", execute_command);
}
```

```
00778:      run_init_process("/sbin/init");
00779:      run_init_process("/etc/init");
00780:      run_init_process("/bin/init");
00781:      run_init_process("/bin/sh");
00782:
00783:      panic("No init found.  Try passing init= option to kernel.");
00784: } ? end init_post ?
```

这几个程序再依次执行也没有故失败，最后打印下面那个错误。

再次关开发板电再上电，在 UBOOT 中：dowload root_yaffs image 下载文件系统。再输入“b”启动。
最后串口可以接收输入，是因为有应用程序在跑了。

```
Please press Enter to activate this console.
starting pid 768, tty '/dev/ttySAC0': '/bin/sh'
#
#
#
```

```
# ps
  PID  Uid        VSZ Stat Command
    1 0         3092 S    init
    2 0              SW< [kthreadd]
    3 0              SWN [ksoftirqd/0]
    4 0              SW< [watchdog/0]
    5 0              SW< [events/0]
    6 0              SW< [khelper]
   55 0              SW< [kblockd/0]
   56 0              SW< [ksuspend_usbd]
   59 0              SW< [khubd]
   61 0              SW< [kseriod]
   73 0              SW  [pdflush]
   74 0              SW  [pdflush]
   75 0              SW< [kswapd0]
   76 0              SW< [aio/0]
  711 0              SW< [mtdblockd]
  746 0              SW< [kmmcd]
  768 0         3096 S    -sh
  769 0         3096 R    ps
#
```

可见第一个应用程序：init 是内核启动的第一个应用程序。
其中的 -sh 就是当前的应用的程序，就是当前接收这个串口。输入字符回显字符。就是 sh 这个程序接收这些东西。

根文件系统源码分析 1

想知道这个文件系统里有哪些东西。就得跟着 init 进程分析，看下这个朝向需要哪些东西。

```
00778:      run_init_process("/sbin/init");
```

文件系统中有很多命令。若很多，则一个一个编译会很麻烦。在嵌入式中有个 busybox.它是一系列命令的组合。编译 busybox 时，得到一个应用程序 busybox,像 ls,cp 等命令就是到这个 busybox
的链接。执行 ls 时，实质上是执行 busybox ls.

```
# ls -l /bin/ls
lrwxrwxrwx    1 1000     1000            7 Jan  6  2010 /bin/ls -> busybox
# ls -l /bin/cp
lrwxrwxrwx    1 1000     1000            7 Jan  6  2010 /bin/cp -> busybox
```

```
# busybox ls
bin         lib         mnt         sbin        usr
dev         linuxrc     proc        sys
etc         lost+found  root        tmp
#
```

这里

```
run_init_process("/sbin/init");
```

这个 init 进程也是到 busybox 的链接。

```
/sbin/init
# ls /sbin/init -l
lrwxrwxrwx    1 1000     1000           14 Jan  6  2010 /sbin/init -> ../bin/busybox
#
```

所以要知道 init 进制做的哪些事，就得看 busybox 源码。

1.建立 busybox 源码 SI 工程，加入全部代码。

```
00024: int cp_main(int argc, char **argv);
00025: int cp_main(int argc, char **argv)
00026: {
```

我们执行 cp 命令时，最终 busybox 会调用这个子函数 int cp_main()

也有一个 init.c 的程序。也有 init_main 函数。

```
00884: int init_main(int argc, char **argv);
00885: int init_main(int argc, char **argv)
00886: {
00887:      struct init_action *a;
00888:      pid_t wpid;
```

（1）.猜测 init 程序要做的事情。

UBOOT 的目标是启动内核，内核的目的是启动应用程序，第一个应用程序是/sbin/init（或其他 init），但最终目标还是启动客户的“用户应用程序”。如手机程序；监控程序等。
这样显然会有一个“配置文件”，在这个文件里指定需要执行哪些不同客户的“应用程序”。
故 init 程序要读取这个：配置文件
再解析这个：配置文件
根据这个“配置文件”：执行客户的用户程序

2.分析 init.c 源码：

```
00904:      signal(SIGHUP, exec_signal);
00905:      signal(SIGQUIT, exec_signal);
00906:      signal(SIGUSR1, shutdown_signal);
00907:      signal(SIGUSR2, shutdown_signal);
00908:      signal(SIGINT, ctrlaltdel_signal);
00909:      signal(SIGTERM, shutdown_signal);
00910:      signal(SIGCONT, cont_handler);
00911:      signal(SIGSTOP, stop_handler);
00912:      signal(SIGTSTP, stop_handler);
```

这些是信号处理函数。里面有 ctrl+alt+del 信号。

```
00921:      console_init();
```

控制台初始化。

```
if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
    printk(KERN_WARNING "Warning: unable to open an initial console.\n");

(void) sys_dup(0);
(void) sys_dup(0);
```

这里初始化这 3 个文件（标准输入、输出、错误）。3 个文件都指向 /dev/console 设备。

1）找到配置文件。/etc/inittab

linux 内核启动

```
run_init_process("/sbin/init");
```

这个应用程序时，没有给参数。

```
if (argc > 1  linxu启动 /sbin/init 时没有给参数，故 argc=1,就执行 else 分枝 parse_inittab()
 && (!strcmp(argv[1], "single") || !strcmp(argv[1], "-s") || LONE_CHAR(argv[1], '1'))
) {
    /* Start a shell on console */
    new_init_action(RESPAWN, bb_default_login_shell, "");
} else {
    /* Not in single user mode -- see what inittab says */

    /* NOTE that if CONFIG_FEATURE_USE_INITTAB is NOT defined,
     * then parse_inittab() simply adds in some default
     * actions(i.e., runs INIT_SCRIPT and then starts a pair
     * of "askfirst" shells */
    parse_inittab();
```

Busybox 调 init_main，init_main 调用 parse_inittab()这个函数。从名字可知是“解析 inittab”表。
分析这个函数（解析 inittab 文件）。

```
00764:      file = fopen(INITTAB, "r");
```

```
00038: #define INITTAB     "/etc/inittab"     /* inittab file location */
```

这就是配置文件。linux 系统中，配置文件一般放在 /etc/ 目录下。

2).分析 /etc/inittab 配置文件。

在 busybox 中有 inittab 说明文档。

```
inittab格式: <id>:<runlevels>:<action>:<process>
```

是启动不同用户的应用程序。则会有：
指定要执行的程序。
何时执行程序。
id 项：/dev/id 用作终端（终端：stdin-printf,stdout-scanf,stderr-err）
会加上一个 /dev 前缀。作为“控制终端”（controlling tty）。
id 可以省略。

runlevels 项：
完全可以忽略掉。
action 项：
指示程序何时执行。
process 项：
是应用程序或脚本。

3）.配置文件结构体和链表

```
00764:      file = fopen(INITTAB, "r");
00765:      if (file == NULL) {
00766:          /* No inittab file -- set up some default behavior */
00767: #endif
00768:          /* Reboot on Ctrl-Alt-Del */
00769:          new_init_action(CTRLALTDEL, "reboot", "");
00770:          /* Umount all filesystems on halt/reboot */
00771:          new_init_action(SHUTDOWN, "umount -a -r", "");
00772:          /* Swapoff on halt/reboot */
00773:          if (ENABLE_SWAPONOFF) new_init_action(SHUTDOWN, "swapoff -a",
00774:          /* Prepare to restart init when a HUP is received */
00775:          new_init_action(RESTART, "init", "");
00776:          /* Askfirst shell on tty1-4 */
00777:          new_init_action(ASKFIRST, bb_default_login_shell, "");
00778:          new_init_action(ASKFIRST, bb_default_login_shell, VC_2);
00779:          new_init_action(ASKFIRST, bb_default_login_shell, VC_3);
00780:          new_init_action(ASKFIRST, bb_default_login_shell, VC_4);
00781:          /* sysinit */
00782:          new_init_action(SYSINIT, INIT_SCRIPT, "");
00783:
00784:          return;
00785: #if ENABLE_FEATURE_USE_INITTAB
00786:      } ?  end if file==NULL ?
```

可见如果 INITTAB 文件不能打开"if(file==NULL)"，则会有“默认配置项”。上面是默认项。

```
00788:      while (fgets(buf, INIT_BUFFS_SIZE, file) != NULL) {
```

如果 file 接收文件 INITTAB 可以找开，则放到一个缓冲区去解析。

```
if (*id == '#' || *id == '\n')
    continue;
```

如果遇到 # 号，则跳过去忽略不执行。

```
if (strncmp(id, "/dev/", 5) == 0)
    id += 5;
```

id 加一个 /dev 前缀。

解析完后，最终执行：

```
new_init_action(a->action, command, id);
```

不论是否有 inittab 这个文件，都会用到这个函数 new_init_action，则需要分析它。

以下面例子来分析：

```
00778:          new_init_action(ASKFIRST, bb_default_login_shell, VC_2);
```

```
new_init_action(ASKFIRST, bb_default_login_shell, VC_2)
#define ASKFIRST       0x004
extern const char bb_default_login_shell[];
在Messages.c中：
```

```
00039: const char bb_default_login_shell[] ALIGN1 = LIBBB_DEFAULT_LOGIN_SHELL;
```

在 Libbb.h 中：

```
01046: #define LIBBB_DEFAULT_LOGIN_SHELL     "-/bin/sh"
```

则 bb_default_login_shell 为 "-/bin/sh"

```
# define VC_2 "/dev/tty2"
```

则：

```
new_init_action(ASKFIRST, bb_default_login_shell, VC_2)
```

变成：

```
new_init_action(0x004, "-/bin/sh", "/dev/tty2")
```

函数原型：

```
static void new_init_action(int action, const char *command, const char *cons)
这个函数参一 action    ：应用程序执行的时机。
        参二 *command ：应用程序或脚本----“"-/bin/sh"”
        参三 *cons     ：这是 id 应用终端---“"/dev/tty2"”
```

```
00698: static void new_init_action(int action, const char *command, const char *cons)
00699: {
00700:     struct init_action *new_action, *a, *last;
00701:
00702:     if (strcmp(cons, bb_dev_null) == 0 && (action & ASKFIRST))
00703:         return;
00704:
00705:     /* Append to the end of the list */
00706:     for (a = last = init_action_list; a; a = a->next) {
00707:         /* don't enter action if it's already in the list,
00708:          * but do overwrite existing actions */
00709:         if ((strcmp(a->command, command) == 0)
00710:          && (strcmp(a->terminal, cons) == 0)
00711:         ) {
00712:             a->action = action;
00713:             return;
00714:         }
00715:         last = a;
00716:     }
00717:
00718:     new_action = xzalloc(sizeof(struct init_action));
00719:     if (last) {
00720:         last->next = new_action;
00721:     } else {
00722:         init_action_list = new_action;
00723:     }
00724:     strcpy(new_action->command, command);
00725:     new_action->action = action;
00726:     strcpy(new_action->terminal, cons);
00727:     messageD(L_LOG | L_CONSOLE, "command='%s' action=%d tty='%s'\n",
00728:         new_action->command, new_action->action, new_action->terminal);
00729: } ? end new_init_action ?
```

里面有一个链表： init_action_list .和结构体：

```
00072: struct init_action {
00073:     struct init_action *next;
00074:     int action;
00075:     pid_t pid;
00076:     char command[INIT_BUFFS_SIZE];
00077:     char terminal[CONSOLE_NAME_SIZE];
00078: };
```

struct init_action *next;为链表用。

action： 执行时机

pid： 进程号

command： 对应执行程序 （-/bin/sh）

terminnal： 终端（/dev/tty2）

从上面可知：new_init_action 这个函数

①.创建结构体 init_action，填充：执行时机，对应执行程序，终端

②.将结构体放入链表 init_action_list

若原来链表中已经有个结构和传进来的 action,*command,*cons 一样就覆盖它。

```
for (a = last = init_action_list; a; a = a->next) {
        /* don't enter action if it's already in the list,
         * but do overwrite existing actions */
        if ((strcmp(a->command, command) == 0)
         && (strcmp(a->terminal, cons) == 0)
        ) {
                a->action = action;
                return;
        }
        last = a;
}
```

若没有，则分配一块内存，相当于创建一个 init_action 结构。将传进来的 action,command,cons 写进链表。

```
00718:        new_action = xzalloc(sizeof(struct init_action));   分配一块内存
00719:        if (last) {
00720:                last->next = new_action;
00721:        } else {
00722:                init_action_list = new_action;
00723:        }
00724:        strcpy(new_action->command, command);
00725:        new_action->action = action;
00726:        strcpy(new_action->terminal, cons);   这里分别对应传进来的 action, terminal, command
00727:        messageD(L_LOG | L_CONSOLE, "command='%s' action=%d tty='%s'\n",
00728:                new_action->command, new_action->action, new_action->terminal);
```

4.从默认的 new_init_action 反推默认的配置文件。

```
00764:        file = fopen(INITTAB, "r");
00765:        if (file == NULL) {
00766:                /* No inittab file -- set up some default behavior */
00767: #endif
00768:                /* Reboot on Ctrl-Alt-Del */
00769:                new_init_action(CTRLALTDEL, "reboot", "");
00770:                /* Umount all filesystems on halt/reboot */
00771:                new_init_action(SHUTDOWN, "umount -a -r", "");
00772:                /* Swapoff on halt/reboot */
00773:                if (ENABLE_SWAPONOFF) new_init_action(SHUTDOWN, "swapoff -a",
00774:                /* Prepare to restart init when a HUP is received */
00775:                new_init_action(RESTART, "init", "");
00776:                /* Askfirst shell on tty1-4 */
00777:                new_init_action(ASKFIRST, bb_default_login_shell, "");
00778:                new_init_action(ASKFIRST, bb_default_login_shell, VC_2);
00779:                new_init_action(ASKFIRST, bb_default_login_shell, VC_3);
00780:                new_init_action(ASKFIRST, bb_default_login_shell, VC_4);
00781:                /* sysinit */
00782:                new_init_action(SYSINIT, INIT_SCRIPT, "");
00783:
00784:                return;
00785: #if ENABLE_FEATURE_USE_INITTAB
00786:        } ?  end if file==NULL ?
```

依据格式分析下面的默认配置。（action 有如下取值。）

```
15 # id => /dev/id, 用作终端: stdin, stdout, stderr: printf ,scanf, err
16 # runlevels : 忽略
17 # action    : 执行时机
18 # <action>: Valid actions include: sysinit, respawn, askfirst, wait, once,
19 #                                  restart, ctrlaltdel, and shutdown.
20 # process   : 应用程序或脚本
```

```
01046: #define LIBBB_DEFAULT_LOGIN_SHELL    "-/bin/sh"
```

```
#define INIT_SCRIPT  "/etc/init.d/rcS"    /* Default sysinit script. */
```

```
/* Reboot on Ctrl-Alt-Del */
new_init_action(CTRLALTDEL, "reboot", "");
这里id为空，runlevels 忽略。action是“CTRLALTDEL”;应用程序是“reboot”。
: : ctrlaltdel:reboot

/* Umount all filesystems on halt/reboot */ 当关闭或重启系统时卸载所有的文件系统
new_init_action(SHUTDOWN, "umount -a -r", "");
::shutdown:umount -a -r

/* Swapoff on halt/reboot */
当PC机有资源不够用时，会将当前内存中的某些应用程序置换到硬盘上。嵌入式中不同这项。
if (ENABLE_SWAPONOFF) new_init_action(SHUTDOWN, "swapoff -a", "");

/* Prepare to restart init when a HUP is received */
new_init_action(RESTART, "init", "");
::restart:init

/* Askfirst shell on tty1-4 */
new_init_action(ASKFIRST, bb_default_login_shell, "");
::askfirst:-/bin/sh

new_init_action(ASKFIRST, bb_default_login_shell, VC_2);
tty2::askfirst:-/bin/sh

new_init_action(ASKFIRST, bb_default_login_shell, VC_3);
tty3::askfirst:-/bin/sh

new_init_action(ASKFIRST, bb_default_login_shell, VC_4);
tty4::askfirst:-/bin/sh

/* sysinit */
new_init_action(SYSINIT, INIT_SCRIPT, "");
::sysinit:/etc/init.d/rcS
```

```
::ctrlaltdel:reboot
::shutdown:umount -a -r
::restart:init
::askfirst:-/bin/sh
tty2::askfirst:-/bin/sh
tty3::askfirst:-/bin/sh
tty4::askfirst:-/bin/sh
::sysinit:/etc/init.d/rcS
```

则如果 UBOOT 没有传配置文件，则会构造出如上这些默认内容。
ENABLE_SWAPONOFF 在 PC 机上如果系统资源不够。会将当前内存中的某些应用程序置放到硬盘上去，
再将新的程序调到内存中。但嵌入式系统一般不会用。

5.可见，parse_ininttab 解析这个 inittab 时，先打开 /etc/inittab 。

再创建很多结构放到链表中。
①.程序调用关系：

```
busybox--> init_main
parse_inittab
file = fopen(INITTAB,"r") //打开配置文件 /etc/inittab
new_init_action // 创建 init_action 结构，填充数据，将结构放到链表init_action_list
// 可见解析inittab后（parse_inittab）得到链表init_action_list，
// 链表里面放有id,执行时机，应用程序。接着再执行。
run_actions(SYSINIT) // 执行，run_actions是复数，故这里是运行 SYSINIT 这一类的动作
waitfor(a, 0);  // 执行应用程序，等待它执行完毕。
run(a)     // 创建process子进程，就是inittab结构中的 process中指定的应用程序（如-/bin/sh）
waitpid(runpid, &status, 0) //等待它结束
delete_init_action(a); // 将这个init_action结构从init_action_list链表中删除掉。
  // 所以用SYSINIT定义的应用程序，执行完一次就扔掉了。
run_actions(WAIT) //与SYSINIT是一样的，只是比SYSINIT后执行。（和上面SYSINIT执行过程一样）
(a->action & (SYSINIT | WAIT | CTRLALTDEL | SHUTDOWN | RESTART))
waitfor(a, 0);
run(a)
waitpid(runpid, &status, 0)
delete_init_action(a);
run_actions(ONCE) //对于“ONCE”这类程序，init进程不会等待它执行完毕（无waitfor函数下调用的waitpid函数）。
run(a);
delete_init_action(a);
接着是“run_actions中的
(a->action & (RESPAWN | ASKFIRST))
”先运行RESPAWN类，再运行ASKFIRST类。
while (1) { 这是个死循环。
```

run_actions(RESPAWN);
if (a->pid == 0) {//只有它们的pid为0时才执行，开始pid是等于0的，若使用了它则给你赋个PID。
a->pid = run(a);
}
run_actions(ASKFIRST);
if (a->pid == 0) { //只有pid为0时才执行。
a->pid = run(a);
打印nPlease press Enter to activate this console.

等待回车：

```
while (read(0, &c, 1) == 1 && c != '\n')
```

不会不往下走
接着创建子进程 BB_EXECVP
}
wpid = wait(NULL); //等待子进程退出。只要任意一个子进程退出时，这个子进程的 pid 就设置为 0。
然后回到 while(1)开始处重新运行，但并不是重新运行所有的子进程，而是运行那个已经退出来的子进程。

```
while (wpid > 0) {
a->pid = 0;     //退出后，就使 pid=0,就退出了上面的循环。
}
}
```

```
else if (a->action & (SYSINIT | WAIT | CTRLALTDEL | SHUTDOWN | RESTART)) {
  waitfor(a, 0);
  delete_init_action(a);
```

run_actions(SYSINIT) 与 run_actions(WAIT) 一样。只是 SYSINIT 先执行。

```
else if (a->action & ONCE) {
    run(a);
    delete_init_action(a);
```

但 ONCE 和 SYSINIT、WAIT 不一样。init 程序不会等待 ONCE 这一类程序执行完毕。

run_actions(ASKFIRST);如果是 ASKFIRST 时，在 run(a)函数中也有区别，会打印信息：

```
    if (a->action & ASKFIRST) {
        static const char press_enter[] ALIGN1 =
#ifdef CUSTOMIZED_BANNER
#include CUSTOMIZED_BANNER
#endif
            "\nPlease press Enter to activate this console. ";

full_write(1, press_enter, sizeof(press_enter) - 1);
while (read(0, &c, 1) == 1 && c != '\n')
```

等待回车

“1”是标准输出。“\nPlease press Enter to activate this console.”再等待“回车键”。
以上便是 SYSINIT ASKFIRST,

什么时候执行是由 action 决定的，action 的状态：sysinit, respawn, askfirst, wait, once, restart, ctrlaltdel, and shutdown.

这些 action 指定了你执行的时机。
还有其他 action 状态（restart,shutdown 等）。一开始时，有设置“信号量”。

```
00904:        signal(SIGHUP, exec_signal);
00905:        signal(SIGQUIT, exec_signal);
00906:        signal(SIGUSR1, shutdown_signal);
00907:        signal(SIGUSR2, shutdown_signal);
00908:        signal(SIGINT, ctrlaltdel_signal);
00909:        signal(SIGTERM, shutdown_signal);
00910:        signal(SIGCONT, cont_handler);
00911:        signal(SIGSTOP, stop_handler);
00912:        signal(SIGTSTP, stop_handler);
```

如上按下 ctrlaltdel 时，内核会给 init 进程发一个信号 SIGINT。init 收到 SIGINT 信号时，就会执行
“ctrlaltdel_signal”信号处理函数。

```
00673: static void ctrlaltdel_signal(int sig ATTRIBUTE_UNUSED)
00674: {
00675:        run_actions(CTRLALTDEL);
00676: }
```

表明会去执行 CTRLALTDEL 这一类的应用程序。

这些状态是什么，busybox 的文档说明中没有，这里只能看源码。如上，run_actions(SYSINIT)首先执行
的是 SYSINIT 这一类，依次如上。（接着执行“run_actions(WAIT)”--"run_actions(ONCE)"）
如上执行关系表。

②.分析 run_action()

```
static void run_actions(int action)
{
        struct init_action *a, *tmp;

        for (a = init_action_list; a; a = tmp) {
                tmp = a->next;
                if (a->action == action) {
                        /* a->terminal of "" means "init's console" */
                        if (a->terminal[0] && access(a->terminal, R_OK | W_OK)) {
                                delete_init_action(a);
                        } else if (a->action & (SYSINIT | WAIT | CTRLALTDEL | SHUTDOWN | RESTART)) {
                                waitfor(a, 0);
                                delete_init_action(a);
                        } else if (a->action & ONCE) {
                                run(a);
                                delete_init_action(a);
                        } else if (a->action & (RESPAWN | ASKFIRST)) {
                                /* Only run stuff with pid==0.  If they have
                                 * a pid, that means it is still running */
                                if (a->pid == 0) {
                                        a->pid = run(a);
                                }
                        }
                }
        } ? end for a=init_action_list;a;... ?
} ? end run_actions ?
```

接收一个参数，上面 run_actions(SYSINIT)传进来一个参数 SYSINIT，则下面从钟表 init_action_list 中取出 a->action 等于这个传进来的 action时，意思即从a->action和传进来的 SYSINIT 相同，则执行。

run_action就从这个链表中将东西取出来

这里是执行。

waitfor()是等待执行完毕的意思

把这个init_action结构从这个“init_action_list”链表中删掉。由run_actions(SYSTEM)中SYSTEM定义的这些程序执行完一次就被扔掉。

run_actions(SYSINIT) ,SYSINIT 故名系统初始化，用 SYSINIT 定义了那些 inittab 结构中的 process 进程。是在最开始时便执行的。并且执行的时候，这个 init 程序会等待这个子进程结束（waitfor(a, 0);）。结束后再将这个子进程从链表中删掉，（delete_init_action(a); 将这个

init_action 结构从 init_action_list 链表中删除掉。）

```
static int waitfor(const struct init_action *a, pid_t pid)
{
    int runpid;
    int status, wpid;

    runpid = (NULL == a)? pid : run(a);
    while (1) {
        wpid = waitpid(runpid, &status, 0);
        if (wpid == runpid)
            break;
        if (wpid == -1 && errno == ECHILD) {
            /* we missed its termination */
            break;
        }
        /* FIXME other errors should maybe trigger an error, but allow
         * the program to continue */
    }
    return wpid;
}
```

创建process子进程，就是inittab结构中的 process中指定的应用程序（如-/bin/sh）.

等待它结束

init.c 中还有如下：

```
BB_EXECVP(cmdpath, cmd);
```

```
#define BB_EXECVP(prog,cmd)     execvp(prog,cmd)
```


这是个系统调用，创建子进程。

总结：最小的根文件系统需要的项：（init 进程需要）

1.打开终端：/dev/console，/dev/NULL
不设置 inittab 格式中的 id(标准输入、输出和标准错误)时，就定位到 /dev/NULL 中去。
2.init 程序本身就是 busybox .
3.需要配置文件：/etc/inittab
4.配置文件若指定了某些应用程序或执行脚本--这些必须存在，不存在会有默认的。
5.应用程序需要 库（fopen、fwrite 等函数需要） 。

根文件系统源码分析 2

最小的根文件系统需要的项：

1.打开终端：/dev/console，/dev/NULL
不设置 inittab 格式中的 id(标准输入、输出和标准错误)时，就定位到 /dev/NULL 中去。
2.init 程序本身就是 busybox .
3.需要配置文件：/etc/inittab
4.配置文件若指定了某些应用程序或执行脚本--这些必须存在，不存在会有默认的。
5.应用程序需要 库 。

1.配置 busybox。

源码的 INSTALL 有说明编译方法。

```
make menuconfig     # This creates a file called ".config"
make                # This creates the "busybox" executable
make install        # or make CONFIG_PREFIX=/path/from/root install
```

最后安装默认会安装到 PC 机上。要装到嵌入式文件系统中，则用后面指定目录的安装方法。

①.首先是给嵌入式系统编译的，所以要有个 交叉编译工具 。

看 Makefile 中的交叉编译的指定。在 linux 中一般交叉编译工具的指定以 CROSS 开头。

```
178: # Architecture as pres
179: UTS_MACHINE := $(ARCH)
```

可以在 Makefile 中直接定义。在配置文件中没法在配置图形项(make menuconfig)中定义，所以就在 Makefile 中定义。直接写成：加上前缀 arm-linux-

```
176: CROSS_COMPILE    ?= arm-linux-
```

或者：make时加上“CROSS_COMPILE=arm-linux-”
make CROSS_COMPILE=arm-linux-

最新版的 busybox 中应该在 make menuconfig 中定义交叉编译工具的项。

②.指定库。

```
[ ] Build BusyBox as a static binary (no shared libs)
```

这是将 BusyBox 编译成一个静态的程序时，那么那些 C 库 也不需要了。除非用户的应用程序指定需要库。这里不选表示用动态的。C 库有两种，glibc 和 uclibc，如果是用 glibc 静态编译时，

会有提示一个警告信息，可能在以后使用时出问题，所以这里不用静态编译。而用动态链接。

```
[*] Build with Large File Support (for accessing files > 2 GB)
```

③.命令补全

```
Busybox Settings  --->
    Busybox Library Tuning  --->
        [*]   Tab completion
```

```
[ ]   History saving
[*]   Tab completion
[ ]     Username completion (NEW)
```

④.压缩命令

```
[*] ar
[*]   Enable support for long filenames (not need for debs)
[*] unzip2
[*] cpio
[ ] cpkg
[ ] cpkg_deb
[*] cunzip
[*]   Uncompress support
[*] czip
[*] cpm2cpio
[*] cpm
[*]   Enable handling of rpms with bzip2-compressed data inside
[*] tar
[*]   Enable archive creation
[*]   Enable -j option to handle .tar.bz2 files
[*]   Enable -a option to handle .tar.lzma files
[*]   Enable -X (exclude from) and -T (include from) options)
[*]   Enable -z option
```

有很多，上面已经默认选择上了。

⑤.模块加载命令，默认也是选择上的。

```
Linux Module Utilities  --->
    [*] insmod
    [*]   Module version checking
    [*]   Add module symbols to kernel symbol table
    [*]   In kernel memory optimization (uClinux only)
    [*]   Enable load map (-m) option
    [*]     Symbols in load map
    [*] rmmod
    [*] lsmod
    [*] Support version 2.6.x Linux kernels
```

默认配置中也是选择好的。

⑦.支持 mdev

（5）Linux System Utilities 选项。

支持 mdev，这可以很方便地构造/dev 目录，并且可以支持热拔插设备。另外，为方便调试，选中 mount、umount 命令，并让 mount 命令支持 NFS（网络文件系统）。

```
Linux System Utilities  --->
    [*] mdev
    [*]   Support /etc/mdev.conf
    [*]     Support command execution at device addition/removal
    [*] mount
    [*]     Support mounting NFS file systems
    [*] umount
    [*]   umount -a option
```

配置中也是默认选择好的。

等等，将需要的命令选择上。

2.编译

make，如遇到错误，就将相应出错的命令去掉，如果需要那个命令，则 google 下错误。

```
ipsvd/tcpudp.c: In function 'tcpudpsvd_main':
ipsvd/tcpudp.c:314: warning: ignoring return value of 'write', declared with attribute
warn_unused_result
make[1]: *** [ipsvd/tcpudp.o] 错误 1
make: *** [ipsvd] 错误 2
```

这是因为 CROSS_COMPILE ?= 没有指定交叉编译工具。而是直接用直接 glibc 编译会有这个错误。

3.安装

make install，一定要指定安装目录，不然就破坏了我们正在使用的系统。命令混淆了。

```
.0# mkdir -p /work/nfs_root/first_fs
```

创建一个文件系统目录。

```
# make CONFIG_PREFIX=/work/nfs_root/first_fs install
```

```
root@book-desktop:/work/system/busybox-1.7.0# mkdir -p /work/nfs_root/first_fs
root@book-desktop:/work/system/busybox-1.7.0# make CONFIG_PREFIX=/work/nfs_root/first_fs i
   /path/from/work/nfs_root/first_fs/bin/addgroup -> busybox
   /path/from/work/nfs_root/first_fs/bin/adduser -> busybox
   /path/from/work/nfs_root/first_fs/bin/ash -> busybox
   /path/from/work/nfs_root/first_fs/bin/cat -> busybox
   /path/from/work/nfs_root/first_fs/bin/catv -> busybox
   /path/from/work/nfs_root/first_fs/bin/chattr -> busybox
   /path/from/work/nfs_root/first_fs/bin/chgrp -> busybox
   /path/from/work/nfs_root/first_fs/bin/chmod -> busybox
   /path/from/work/nfs_root/first_fs/bin/chown -> busybox
```

```
book@book-desktop:/work/nfs_root/first_fs# ls -l
total 12
drwxr-xr-x 2 book book 4096 2010-11-26 18:27 bin
lrwxrwxrwx 1 book book   11 2010-11-26 18:27 linuxrc -> bin/busybox
drwxr-xr-x 2 book book 4096 2010-11-26 18:27 sbin
drwxr-xr-x 4 book book 4096 2010-11-26 18:27 usr
book@book-desktop:/work/nfs_root/first_fs#
```

```
book@book-desktop:/work/nfs_root/first_fs# ls bin/ls -l
lrwxrwxrwx 1 book book 7 2010-11-26 18:27 bin/ls -> busybox
book@book-desktop:/work/nfs_root/first_fs#
```

4.创建最简单的文件系统。

①.创建 /dev/console，/dev/NULL

PC 机上的这两个文件：

```
root@book-desktop:/work/system/busybox-1.7.0# ls /dev/console /dev/null -l
crw------- 1 root root 5, 1 2011-03-06 11:11 /dev/console
crw-rw-rw- 1 root root 1, 3 2011-03-06 11:11 /dev/null
```

c 表示是字符设备，5 表示它的主设备号，1 是次设备号

在 first_fskh 创建 dev 目录，并创建这两个 文件设备。

```
:/first_fs# mkdir dev
:/first_fs# cd dev/
```

```
/first_fs/dev# mknod console c 5 1
/first_fs/dev# mknod null c 1 3
```

用 mknod 创建 console ，c 表示字符型设备，5 是主设备号，1 是次设备号。

②.init 程序本身就是 busybox，所以这一步完成。

③.创建 /etc/inittab 配置文件。

若不构造，则使用默认的配置项：

```
::ctrlaltdel:reboot
::shutdown:umount -a -r
::restart:init
::askfirst:-/bin/sh
tty2::askfirst:-/bin/sh
tty3::askfirst:-/bin/sh
tty4::askfirst:-/bin/sh
::sysinit:/etc/init.d/rcS
```

不需要这个，所以自己写。只写一条 ::askfirst:-/bin/sh

先在 first_fs 目录下创建 etc 目录，再在 etc 下创建 inittab 文件。

```
console::askfirst:-/bin/sh
```

表示它里面只执行一个 askfirst 一个/bin/sh 程序，/bin/sh 这个程序的标准输入、输出和标准错误都

定位到“/dev/console”里面去。

④.安装 glibc 库

```
mkdir -p /work/nfs_root/fs_mini/lib
cd /work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib
cp *.so* /work/nfs_root/fs_mini/lib -d
```

将所有 .so 文件拷贝过来即可。-d 是本来是个链接文件，则也保持是个链接文件。不然就拷贝成了真正的文件，这样会很大。

```
root@book-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib# ls
crt1.o                  libcrypt.a              libm-2.3.6.so           libpng12.a              libstdc++.la
crti.o                  libcrypt.so             libm.a                  libpng12.so             libstdc++.so
crtn.o                  libcrypt.so.1           libmcheck.a             libpng12.so.0           libstdc++.so.6
gconv                   libc.so                 libmemusage.so          libpng12.so.0.1.2.23    libstdc++.so.6.0.3
gcrt1.o                 libc.so.6               libm.so                 libpng.a                libsupc++.a
ld-2.3.6.so             libc.so_orig            libm.so.6               libpng.so               libsupc++.la
ld-linux.so.2           libdl-2.3.6.so          libnsl-2.3.6.so         libpng.so.3             libthread_db-1.0.so
ldscripts               libdl.a                 libnsl.a                libpng.so.3.1.2.23      libthread_db.so
libanl-2.3.6.so         libdl.so                libnsl.so               libpthread-0.10.so      libthread_db.so.1
libanl.a                libdl.so.2              libnsl.so.1             libpthread.a            libts-0.0.so.0
libanl.so               libe2p.a                libnss_compat-2.3.6.so  libpthread_nonshared.a  libts-0.0.so.0.1.0
libanl.so.1             libe2p.so               libnss_compat.so        libpthread.so           libts.la
libblkid.a              libe2p.so.2             libnss_compat.so.2      libpthread.so.0         libts.so
libblkid.so             libe2p.so.2.3           libnss_dns-2.3.6.so     libpthread.so_orig      libutil-2.3.6.so
libblkid.so.1           libext2fs.a             libnss_dns.so           libresolv-2.3.6.so      libutil.a
libblkid.so.1.0         libext2fs.so            libnss_dns.so.2         libresolv.a             libutil.so
libBrokenLocale-2.3.6.so libext2fs.so.2         libnss_files-2.3.6.so   libresolv.so            libutil.so.1
libBrokenLocale.a       libext2fs.so.2.4        libnss_files.so         libresolv.so.2          libuuid.a
libBrokenLocale.so      libg.a                  libnss_files.so.2       librt-2.3.6.so          libuuid.so
libBrokenLocale.so.1    libgcc_s.dir            libnss_hesiod-2.3.6.so  librt.a                 libuuid.so.1
libbsd-compat.a         libgcc_s.so             libnss_hesiod.so        librt.so                libuuid.so.1.2
libc-2.3.6.so           libgcc_s.so.1           libnss_hesiod.so.2      librt.so.1              libz.so
libc.a                  libiberty.a             libnss_nis-2.3.6.so     libSegFault.so          libz.so.1
libc_nonshared.a        libieee.a               libnss_nisplus-2.3.6.so libss.a                 libz.so.1.2.3
libcom_err.a            libjpeg.a               libnss_nisplus.so       libss.so                Mcrt1.o
libcom_err.so           libjpeg.la              libnss_nisplus.so.2     libss.so.2              pkgconfig
libcom_err.so.2         libjpeg.so              libnss_nis.so           libss.so.2.0            Scrt1.o
libcom_err.so.2.1       libjpeg.so.62           libnss_nis.so.2         libstdc++.a
libcrypt-2.3.6.so       libjpeg.so.62.0.0       libpcprofile.so         libstdc++.dir
root@book-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib#
```

这是所有的库。其中 .a 的文件是静态库，我们只要动态库。

```
libnss_dns.so -> libnss_dns.so.2
```

libnss_dns.so 是一个链接文件，链接到 libnss_dns.so.2 若拷贝时不加 -d ，则会 libnss_dns.so 拷贝成 libnss_dns.so.2 的内容。这个库就会显得很大。

```
/gcc-3.4.5-glibc-2.3.6/arm-linux/lib# mkdir /work/nfs_root/first_fs/lib
/gcc-3.4.5-glibc-2.3.6/arm-linux/lib# cp *.so* /work/nfs_root/first_fs/lib/ -d
/gcc-3.4.5-glibc-2.3.6/arm-linux/lib#
```

拷贝动态库文件。

```
root@book-desktop:/work/nfs_root/first_fs# ls dev/ lib/ etc/
dev/:
console  null

etc/:
inittab

lib/:
ld-2.3.6.so              libc.so_orig       libnsl.so              libpcprofile.so       libss.so.2.0
ld-linux.so.2            libdl-2.3.6.so     libnsl.so.1            libpng12.so           libstdc++.so
libanl-2.3.6.so          libdl.so           libnss_compat-2.3.6.so libpng12.so.0         libstdc++.so.6
libanl.so                libdl.so.2         libnss_compat.so       libpng12.so.0.1.2.23  libstdc++.so.6.0.3
libanl.so.1              libe2p.so          libnss_compat.so.2     libpng.so             libthread_db-1.0.so
libblkid.so              libe2p.so.2        libnss_dns-2.3.6.so    libpng.so.3           libthread_db.so
libblkid.so.1            libe2p.so.2.3      libnss_dns.so          libpng.so.3.1.2.23    libthread_db.so.1
libblkid.so.1.0          libext2fs.so       libnss_dns.so.2        libpthread-0.10.so    libts-0.0.so.0
libBrokenLocale-2.3.6.so libext2fs.so.2     libnss_files-2.3.6.so  libpthread.so         libts-0.0.so.0.1.0
libBrokenLocale.so       libext2fs.so.2.4   libnss_files.so        libpthread.so.0       libts.so
libBrokenLocale.so.1     libgcc_s.so        libnss_files.so.2      libpthread.so_orig    libutil-2.3.6.so
libc-2.3.6.so            libgcc_s.so.1      libnss_hesiod-2.3.6.so libresolv-2.3.6.so    libutil.so
libcom_err.so            libjpeg.so         libnss_hesiod.so       libresolv.so          libutil.so.1
libcom_err.so.2          libjpeg.so.62      libnss_hesiod.so.2     libresolv.so.2        libuuid.so
libcom_err.so.2.1        libjpeg.so.62.0.0  libnss_nis-2.3.6.so    librt-2.3.6.so        libuuid.so.1
libcrypt-2.3.6.so        libm-2.3.6.so      libnss_nisplus-2.3.6.so librt.so             libuuid.so.1.2
libcrypt.so              libmemusage.so     libnss_nisplus.so      librt.so.1            libz.so
libcrypt.so.1            libm.so            libnss_nisplus.so.2    libSegFault.so        libz.so.1
libc.so                  libm.so.6          libnss_nis.so          libss.so              libz.so.1.2.3
libc.so.6                libnsl-2.3.6.so    libnss_nis.so.2        libss.so.2
root@book-desktop:/work/nfs_root/first_fs#
```

```
root@book-desktop:/work/nfs_root/first_fs# ls
bin  dev  etc  lib  linuxrc  sbin  usr
```

以上便是最小的根文件系统。

⑤.将这个文件系统烧到 FLASH 上。

制作 yaffs 文件系统映象。yaffs1 是给小页的 NAND FLASH 用的，是每页 512 字节。现在大 NAND 是 2048 字节。得用 yaffs2 .

```
yaffs_source_util_larger_small_page_nand.tar.bz2
```

这个 yaffs 工具修改过代码，可以支持小页和大页 NAND。

解压出来是

```
Development_util_ok
```

```
root@book-desktop:/work/system/Development_util_ok/yaffs2/utils# ls
Makefile  mkyaffs2image.c  mkyaffsimage.c  nand_ecc.c  yaffs_packedtags1.c
root@book-desktop:/work/system/Development_util_ok/yaffs2/utils# make
```

在上面这个目录执行 make。

```
root@book-desktop:/work/system/Development_util_ok/yaffs2/utils# ls
Makefile         mkyaffs2image.o  mkyaffsimage.o  yaffs_ecc.c          yaffs_packedtags1.o  yaffs_tagsvali
mkyaffs2image    mkyaffsimage     nand_ecc.c      yaffs_ecc.o          yaffs_packedtags2.c  yaffs_tagsvali
mkyaffs2image.c  mkyaffsimage.c   nand_ecc.o      yaffs_packedtags1.c  yaffs_packedtags2.o
root@book-desktop:/work/system/Development_util_ok/yaffs2/utils#
```

生成了 mkyaffs2image 这个工具。将这个工具直接拷贝到这台 PC 机的 /usr/local/bin 目录下。再加上可执行属性。这样便可直接使用了。

```
:#
:# cp mkyaffs2image /usr/local/bin/
:# chmod +x /usr/local/bin/mkyaffs2image
:#
```

接着制作这个根文件系统：first_fs.yaffs2

```
root@book-desktop:/work/nfs_root# mkyaffs2image
mkyaffs2image: image building tool for YAFFS2 built Mar  7 2011
usage: mkyaffs2image dir image_file [convert]
           dir        the directory tree to be converted
           image_file the output file to hold the image
           'convert'  produce a big-endian image from a little-endian machine
root@book-desktop:/work/nfs_root# mkyaffs2image first_fs first_fs.yaffs2
```

```
Operation complete.
374 objects in 9 directories
5010 NAND pages
root@book-desktop:/work/nfs_root# ls first_fs.
first_fs.jffs2   first_fs.yaffs2
root@book-desktop:/work/nfs_root# ls first_fs.yaffs2
```

下载文件系统时输入“y”之后用 dnw 工具下载根文件系统到开发板。

```
[y] Download root_yaffs image
```

烧写完最小根文件系统后，输入“b”启动系统。

以上便是最小的根文件系统了，但是用 ps 时会提示没有 /proc 目录的错误。则要完善。

```
# ps
  PID  Uid        VSZ Stat Command
ps: can't open '/proc': No such file or directory
#
```



```
# mkdir proc
# ps
  PID  Uid        VSZ Stat Command
#
```

5.完善根文件系统。

①.挂载虚拟的根文件系统。

```
first_fs# mkdir proc
```

在内核中，当前有哪些应用程序在跑，这些信息如何收集。是内核提供了一个虚拟的文件系统，叫 proc 文件系统。

将这个内核提供的虚拟文件系统 proc 挂载一下。

```
# mount -t proc none /proc
# ps
  PID  Uid        VSZ Stat Command
    1 0         3092 S    init
    2 0              SW< [kthreadd]
    3 0              SWN [ksoftirqd/0]
    4 0              SW< [watchdog/0]
    5 0              SW< [events/0]
    6 0              SW< [khelper]
   55 0              SW< [kblockd/0]
   56 0              SW< [ksuspend_usbd]
   59 0              SW< [khubd]
   61 0              SW< [kseriod]
   73 0              SW  [pdflush]
   74 0              SW  [pdflush]
   75 0              SW< [kswapd0]
   76 0              SW< [aio/0]
  711 0              SW< [mtdblockd]
  746 0              SW< [kmmcd]
  764 0         3096 S    -sh
  773 0         3096 R    ps
# 
```

这时 ps 命令便去 /proc 目录看下有哪些程序，看里面有什么内容。

```
# cd /proc/
# ls
1            746          diskstats    locks        sys
2            75           driver       meminfo      sysrq-trigger
3            76           execdomains  misc         sysvipc
4            764          fb           modules      timer_list
5            774          filesystems  mounts       tty
55           asound       fs           mtd          uptime
56           buddyinfo    interrupts   net          version
59           bus          iomem        partitions   vmstat
6            cmdline      ioports      scsi         yaffs
61           cpu          irq          self         zoneinfo
711          cpuinfo      kallsyms     slabinfo
73           crypto       kmsg         stat
74           devices      loadavg      swaps
#
```

1 表示上面 PID 为 1 的程序，是 init 这个程序。

```
# cd 1
# ls -l fd
lrwx------    1 0        0               64 Jan  1 00:02 0 -> /dev/console
lrwx------    1 0        0               64 Jan  1 00:02 1 -> /dev/console
lrwx------    1 0        0               64 Jan  1 00:02 2 -> /dev/console
# 
```

里面的 fd 就指向了 /dev/console,三个分别是标准输入、输出和标准错误。

上面这个 proc 制作的过程可以在运行了最小根文件系统的单板上如上步骤制作。但可以直接制作好烧到单板上。就是在用

mkyaffs2image 工具制作文件系统时，用上面的步骤建立这个 Proc 目录，并将 mount 过程自动执行（加进 init/）。

②.在文件系统加 /proc 目录，修改配置文件 ，

使内核本身提供的虚拟文件系统 proc 挂接到/proc 目录下。

```
  1 console::askfirst:-/bin/sh
  2 ::sysinit:/etc/init.d/rcS
~
```

加一个脚本“rcS”。

```
/first_fs# mkdir etc/init.d
/first_fs# vim etc/init.d/rcS
```

```
File  Edit  View  Terminal  Help
  1 mount -t proc none /proc
~
```

```
/first_fs# chmod +x etc/init.d/rcS
```

这样重新烧这个文件系统到开发板时，一起动就会直接执行 mount -t proc none /proc
若还想在开始启动时执行其他命令，则也可以加到 rcS 脚本中去。

除了 mount -t proc none /proc 外，还有另一个方法 mount -a.它是去读 /etc/fstab，根据这个 fstab 文件里的内容来挂载
文件系统。将 etc/init.d/rcS 中修改如下：

```
1 #mount -t proc none /proc
2 mount -a
```

这个 mount -a 依赖 etc/fstab 中的内容。在 busybox 的源码中有一个 fstab 的说明：

```
# device    mount-point    type    options      dump  fsck order
proc         /proc         proc    defaults      0     0
tmpfs        /tmp          tmpfs   defaults      0     0
```

/etc/fstab 文件被用来定义文件系统的“静态信息”，这些信息被用来控制 mount 命令的行为。文件中各字段意义如下。

① device：要挂接的设备。

比如/dev/hda2、/dev/mtdblock1 等设备文件；也可以是其他格式，比如对于 proc 文件系统这个字段没有意义，可以是任意值；对于 NFS 文件系统，这个字段为<host>:<dir>。

② mount-point：挂接点。

③ type：文件系统类型。

比如 proc、jffs2、yaffs、ext2、nfs 等；也可以是 auto，表示自动检测文件系统类型。

④ options：挂接参数，以逗号隔开。

对于虚拟文件系统 proc 格式中的“device”处的名字可以随便写，上面写的是“proc”。
mount-point ：是挂载到哪里去。
文件系统类型“type”是“proc”虚拟文件系统。
“options”是参数。

```
root@book-desktop:/work/nfs_root/first_fs# cat etc/init.d/rcS
#mount -t proc none /proc
mount -a
root@book-desktop:/work/nfs_root/first_fs# cat etc/fstab
#device         mount-point             type    options         dump    fsck    orde
proc                    /proc           proc    defaults         0       0
root@book-desktop:/work/nfs_root/first_fs#
```

脚本 rcS 中用命令 mount -a 来读取文件 etc/fstab。根据 fstab 的指示去挂载文件系统。

这时可以再 mkyaffs2image first_fs first_fs.yaffs2 制作一次文件系统。烧到开发板。
在单板上执行如下：

```
cat /proc/mounts
```

```
# cat /proc/mounts
rootfs / rootfs rw 0 0
/dev/root / yaffs rw 0 0
proc /proc proc rw 0 0
```

可以查看当前挂载的文件系统。

6.完善 dev 目录。

```
# ls /dev
console  null
# 
```

dev 目录下对应那些设备和驱动。有成千上百驱动，一个个创建会很麻烦。用 udev 机制。udev 是 linux 下自动创建 dev 目录下的设备节点。busybox 中有个简化版本 mdev（udev 的简化版本）。

busybox 目录下有一个 mdev.txt 有说明要做如下 6 项。

```
$ mount -t tmpfs mdev /dev                    /* 使用内存文件系统，减少对 Flash 的读
$ mkdir /dev/pts                          /* devpts 用来支持外部网络连接(telnet)的虚拟
$ mount -t devpts devpts /dev/pts
$ mount -t sysfs sysfs /sys                       /* mdev 通过 sysfs 文件系统获得设备信
$ echo /bin/mdev > /proc/sys/kernel/hotplug   /* 设置内核，当有设备拔插时调用/bin/
程序 */
$ mdev -s                                     /* 在/dev 目录下生成内核支持的所有设备的结
```

[1] mount -t sysfs sysfs /sys 首先挂载一个文件系统到 sys 目录中去。

[2] echo /bin/mdev > /proc/sys/kernel/hotplug

[3] mdev -s

[4] mount -t tmpfs mdev /dev 还要挂载一个文件系统到 dev 目录中去。

[5] mkdir /dev/pts

[6] mount -t devpts devpts /dev/pts

在根目录下：

```
root@book-desktop:/work/nfs_root/first_fs# mkdir sys
root@book-desktop:/work/nfs_root/first_fs# ls
bin  dev  etc  lib  linuxrc  proc  sbin  sys  usr
root@book-desktop:/work/nfs_root/first_fs# 
```

在如下两个文件中加如下内容。

① etc/fstab。

```
# device    mount-point     type    options         dump  fsck order
proc        /proc           proc    defaults        0     0
tmpfs       /tmp            tmpfs   defaults        0     0
sysfs       /sys            sysfs   defaults        0     0
tmpfs       /dev            tmpfs   defaults        0     0
```

第1、4步

② etc/init.d/rcS：加入下面几行。

```
mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

其他步骤

mount -a 就根据上面 /etc/fstab 中的内容挂接好的文件系统。
第 1、4 步在 etc/fstab 中添加。
第 2、3、5、6 步在 etc/init.d/rcS 中添加。

先做第一步和第四步：etc/fstab 中添加

```
1 #device     mount-point     type    options     dump    fsck    order
2 proc            /proc      proc    defaults     0       0
3 sysfs           /sys       sysfs   defaults     0       0
4 tmpfs           /dev       tmpfs   defaults     0       0
```

第1、4步

etc/init.d/rcS 中添加：

```
1 #mount -t proc none /proc
2 mount -a
3 mkdir /dev/pts
4 mount -t devpts devpts /dev/pts
5 echo /sbin/mdev > /proc/sys/kernel/hotplug
6 mdev -s
```

当自动加载某个驱动，或热挺拔 U 盘等。内核就会调用 /proc/sys/kernel/hotplug
这个 hotplug 指向 /sbin/mdev，这个程序就会自动创建那些设备节点。
mdev -s： -s 是指一开始就先将原先内核现有的那些驱动设备节点都创建出来。

再次重新制作 文件系统，烧写到开发板：

```
# ls
bin         etc         linuxrc     proc        sys
dev         lib         lost+found  sbin        usr
# ls dev
console         ptyrf           tty17           ttyq7
dsp             ptys0           tty18           ttyq8
event0          ptys1           tty19           ttyq9
fb0             ptys2           tty2            ttyqa
full            ptys3           tty20           ttyqb
kmem            ptys4           tty21           ttyqc
kmsg            ptys5           tty22           ttyqd
loop0           ptys6           tty23           ttyqe
loop1           ptys7           tty24           ttyqf
loop2           ptys8           tty25           ttyr0
loop3           ptys9           tty26           ttyr1
loop4           ptysa           tty27           ttyr2
loop5           ptysb           tty28           ttyr3
loop6           ptysc           tty29           ttyr4
loop7           ptysd           tty3            ttyr5
mem             ptyse           tty30           ttyr6
mice            ptysf           tty31           ttyr7
mixer           ptyt0           tty32           ttyr8
mouse0          ptyt1           tty33           ttyr9
mtd0            ptyt2           tty34           ttyra
```

这些就是 mdev 自动创建的。

```
# cat /proc/mounts
rootfs / rootfs rw 0 0
/dev/root / yaffs rw 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
tmpfs /dev tmpfs rw 0 0
devpts /dev/pts devpts rw 0 0
#
```

以上最小的文件系统基本完善了。

想要用其他的文件系统格式：jffs2

用 jffs2 文件系统映象。这是一般用在 NORFLASY 上的，但也可以用在 NAND 上。

①.配置

```
:/work/GUI/xwindow/X/deps# tar xzf zlib-1.2.3.tar.gz
:/work/GUI/xwindow/X/deps# cd zlib-1.2.3
:/work/GUI/xwindow/X/deps/zlib-1.2.3#
```

在 linux 下很多程序都要先配置，才能编译。

```
:/work/GUI/xwindow/X/deps/zlib-1.2.3# ./configure --shared --prefix=/usr
```

```
--shared 是会编译出它的动态库
--prefix 是指安装到哪个目录
```

②.编译和安装

安装是将它安装到系统目录中去的。但如果某个程序是 交叉编译 的，则不要安装在系统目录下。

这样会破坏系统。

zlib 是个压缩库。jffas2 文件系统是个压缩的文件系统。所以 制作这个压缩的文件系统时，需要 zlib 这个包。把这个包安装到系统应该是可以的不会破坏当前的操作系统，若是交叉编译时就不要安装到系统目录中，得别建目录安装。

用 ./configure 配置时，产生的 Makefile 中是 gcc

```
CC=gcc
```

③.安装 jffas2 压缩文件系统的工具：

```
:/work/tools# tar jxf mtd-utils-05.07.23.tar.bz2
:/work/tools# cd mtd-utils-05.07.23
:/work/tools/mtd-utils-05.07.23# cd util/
:/work/tools/mtd-utils-05.07.23/util# make
```

是进入 util 目录下 make 编译。
接着安装 make install 安装到系统上。

④.制作

```
:/work/nfs_root# mkfs.jffs2 -n -s 2048 -e 128KiB -d first_fs -o first_fs.jffas2
```

-n 表示不要在每个擦除块上都加上清除标志。
-s 指明一页的大小是多少。
NAND每页是2048字节.
-e 是一个擦除块的大小 （单位 KiB）
NAND是每块128KiB
-d 是制作哪个目录为文件系统
-o 是输入文件系统

压缩的文件系统几乎比 yaffs 文件系统小一半（4M 多点）。使用不同的 flash 则相应的参数需要修改。

重新烧写

```
[j] Download root_jffs2 image
```

重新启动后还是以 yaffs 文件系统来挂接的。

```
VFS: Mounted root (yaffs filesystem).
```

这显然不对。得指定下根文件系统的类型。不能自动识别则在命令行中强制指定。再启动。

```
# set bootargs noinitrd console=ttySAC0 root=/dev/mtdblock1 rootfstype=jffs2
# saveenv
```

```
OpenJTAG> set bootargs noinitrd root=/dev/mtdblock3 rootfstype=jffs2 init=/linuxrc console=ttySAC0
OpenJTAG> save
```

重启后便以 jffs2 来挂载了

```
VFS: Mounted root (jffs2 filesystem).
```

网络文件系统。这个文件系统放在服务器上，内核启动时真识别服务器上这个目录，将这个目录当成 根文件系统。

这样便不需要每次烧写。

```
# ifconfig eth0 up
eth0: link down
# ifconfig
eth0      Link encap:Ethernet  HWaddr 08:90:90:90:90:90
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:51 Base address:0xa000
```

接上网线，配置上 IP，开发板之前有一个根文件系统，配置 IP 后要与服务器可通。

挂接 NFS 要求：
服务器允许那个目录被挂接。--启动 nfs 服务（/etc/exports 在这个配置文件中定义这个目录）。

单板去挂接。

①.在 PC 上开启 nfs 服务并测试。

从 flash 上启动根文件系统，再用命令挂接 NFS 。
在虚拟机上启动 nfs 服务（配置文件在 /etc/exports ），在这个服务的配置文件中，加上要被单板挂接的目录：
这里让单板挂接的目录是“/work/nfs_root/first_fs”

```
/work/nfs_root/first_fs# vim /etc/exports
```



```
 10 #
 11 /work/nfs_root *(rw,sync,no_root_squash)
 12 /work/nfs_root/first_fs    *(rw,sync,no_root_squash)
~
```

再启动 PC 上这个服务：

```
root@book-desktop:/work/nfs_root/first_fs# /etc/init.d/nfs-kernel-server restart
 * Stopping NFS kernel daemon                                                    [ OK
 * Unexporting directories for NFS kernel daemon...                              [ OK
 * Exporting directories for NFS kernel daemon...
exportfs: /etc/exports [1]: Neither 'subtree_check' or 'no_subtree_check' specified for export "*:/work/nfs_root".
  Assuming default behaviour ('no_subtree_check').
  NOTE: this default has changed since nfs-utils version 1.0.x

exportfs: /etc/exports [2]: Neither 'subtree_check' or 'no_subtree_check' specified for export "*:/work/nfs_root/firs
fs".
  Assuming default behaviour ('no_subtree_check').
  NOTE: this default has changed since nfs-utils version 1.0.x

                                                                               [ OK
 * Starting NFS kernel daemon                                                  [ OK
root@book-desktop:/work/nfs_root/first_fs#
```

此时，在 PC 机上测试下是否可以自己挂接自己：

```
root@book-desktop:/work/nfs_root/first_fs# mount -t nfs 172.16.1.1:/work/nfs_root/first_fs /
root@book-desktop:/work/nfs_root/first_fs# ls /mnt/
bin  dev  etc  lib  linuxrc  proc  sbin  sys  usr
root@book-desktop:/work/nfs_root/first_fs#
```

②.单板挂接

```
# mkdir /mnt
# mount -t nfs -o nolock 192.168.1.19:/work/nfs_root/first_fs /mnt
# ls /mnt
bin     etc     linuxrc  sbin    usr
dev     lib     proc     sys
#
```

在单板上创建一个挂接目录，再挂接。这样在服务器上改变那个目录下的文件时，单板上挂接的这个目录里也会改变。
这种方法是开发板启动之后去挂载的。
直接让开发板从 nfs 启动。修改命令行参数。

```
OpenJTAG> print
bootcmd=nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0
baudrate=115200
ethaddr=08:00:3e:26:0a:5b
ipaddr=192.168.7.17
serverip=192.168.7.11
netmask=255.255.255.0
mtdids=nand0=nandflash0
mtdparts=mtdparts=nandflash0:256k@0(bootloader),128k(params),2m(kernel),-(root)
bootdelay=10
bootargs=noinitrd root=/dev/mtdblock3 rootfstype=jffs2 init=/linuxrc console=ttySAC0
stdin=serial
stdout=serial
stderr=serial
partition=nand0,0
mtddevnum=0
mtddevname=bootloader

Environment size: 461/131068 bytes
OpenJTAG>
```

上面是从“/dev/mtdblock3” 启动的。

```
bootargs=noinitrd root=/dev/mtdblock3 rootfstype=jffs2 init=/linuxrc console=ttySAC0
```

要求：
服务 IP、目录。
开发板 IP。
在 linux 内核的 Documention\nfsroot.txt 中有说明用法。

```
nfsroot=[<server-ip>:]<root-dir>[,<nfs-options>]
ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>
noinitrd root=/dev/nfs nfsroot=[<server-ip>:]<root-dir>[,<nfs-options>] ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>

[<server-ip>:] 服务器IP。
<root-dir> 服务器上哪个目录设置成被单板挂载。
[<nfs-options>] 用中括号表示的参数可以省略。尖括号的不可省略。
<client-ip> 表示单板的IP。
<server-ip> 服务器IP。
<gw-ip> 网关，单板和服务器同一网段。
<netmask> 子网掩码
<hostname> 不关心这个，不要。
<device> 网卡，如 eth0\eth1
<autoconf> 自动配置，这个不需要。写成 off
```

```
set bootargs noinitrd root=/dev/nfs nfsroot=172.16.1.11:/work/nfs_root/romfs
```

```
ip=172.16.1.100:172.16.1.11:172.16.1.11:255.255.255.0::eth0:off init=/linuxrc console=ttySAC0
```

```
这是原来的启动参数：noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0 mem=64M
```

配置在单板上，再"save"+"boot"

```
bootargs=noinitrd root=/dev/nfs nfsroot=192.168.1.19:/work/nfs_root/first_fs ip=192.168.1.11:192.168.1.19:192.168.1.1:255.25
:eth0:off init=/linuxrc console=ttySAC0

Environment size: 548/131068 bytes
OpenJTAG> boot
```

设置后重启单板。

```
eth0: link up, 100Mbps, full-duplex, lpa 0x45E1
IP-Config: Complete:
      device=eth0, addr=192.168.1.11, mask=255.255.255.0, gw=192.168.1.1,
     host=192.168.1.11, domain=, nis-domain=(none),
     bootserver=192.168.1.19, rootserver=192.168.1.19, rootpath=
Looking up port of RPC 100003/2 on 192.168.1.19
Looking up port of RPC 100005/1 on 192.168.1.19
VFS: Mounted root (nfs filesystem).
Freeing init memory: 136K
init started: BusyBox v1.7.0 (2010-11-26 18:26:44 CST)
starting pid 766, tty '': '/etc/init.d/rcS'

Please press Enter to activate this console.
starting pid 771, tty '/dev/console': '/bin/sh'
#
# ls
bin         etc         linuxrc     sbin        test.txt
dev         lib         proc        sys         usr
#
```

这样单板就挂接好了 nfs 网络文件系统了。在服务器上写一个程序，用 arm-linux-gcc 编译，放到这个
nfs 文件系统上。在单板上便可以运行。

```
svc: failed to register lockdv1 RPC service (errno 111).
lockd_up: makesock failed, error=-111
mount: mounting 10.61.0.50:/root_nfs on /mnt failed: Connection refused
```

网上找了一下，再输入：

```
mount -t nfs    -o nolock 10.61.0.50:/root_nfs /mnt/
```

这样就好了。
这样可以挂载本机的 nfs，但是在开发板的 bootloader 命令行中输入的参数后确挂载不了，如是我想到干脆把 nfs 挂载成开发板的根文件系统。输入：

```
mount -t nfs -o remount,rw,nolock    10.61.0.50:/root_nfs    /
```

这样居然就成功的挂载了 nfs 作为开发板的文件系统了