

LINUX内核配置编译

一、linux内核配置

1、配置流程概述

linux内核功能非常多，支持多种架构，因此在编译之前需要通过配置来选中和裁剪内核，将需要的文件选中，在后来的编译过程中会自动根据配置的结果来编译和链接选中的文件。配置方式有通过默认配置make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- LOADADDR=0x8000 XXX_defconfig，或者配置工具如make menuconfig来配置，最终的主要目的是得到一个.config文件和一些根据这个文件得到的附带文件如makefile调用的auto.conf，这个文件和.config文件内容几乎一样。还可以将以前配置得到的.config文件保存起来然后直接复制给.config。

2、.config

配置的最终目的是得到.config文件和其他一些相应的文件，.config文件指导接下来的编译过程，比如将哪些功能相关的代码文件编译进内核，哪些编译成模块，哪些不编译。下面是配置生成的.config文件的一个片段。

```
CONFIG_INET_TUNNEL=m
CONFIG_INET_XFRM_MODE_TRANSPORT=y
CONFIG_INET_XFRM_MODE_TUNNEL=y
CONFIG_INET_XFRM_MODE_BEET=y
CONFIG_INET_LRO=y
CONFIG_INET_DIAG=y
CONFIG_INET_TCP_DIAG=y
# CONFIG_INET_UDP_DIAG is not set
# CONFIG_TCP_CONG_ADVANCED is not set
CONFIG_TCP_CONG_CUBIC=y
CONFIG_DEFAULT_TCP_CONG="cubic"
# CONFIG_TCP_MD5SIG is not set
CONFIG_IPV6=m
```

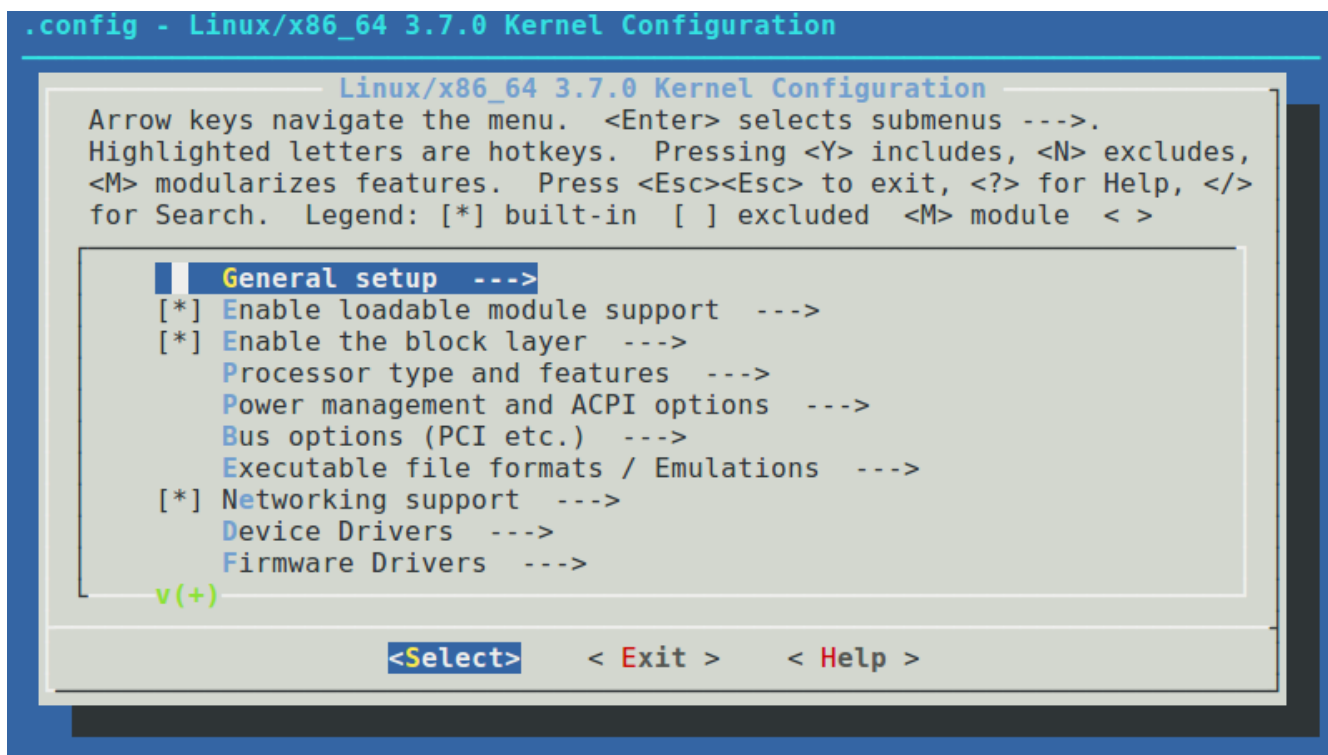
上面是.config文件中的一个片段，这个片段展示了.config文件的典型配置模式。

- (1) #开头的是注释：比如# CONFIG_INET_UDP_DIAG is not set，它说明了INET_UDP_DIAG功能没有配置进内核，所以后面的编译不会把它相关的代码文件编译。
- (2) 配置为y：说明该子系统会被编译和静态链接到内核镜像中去。
- (3) 配置为m：说明该子系统会被编译成模块，当内核启动后可以通过工具来加载它使用。
- (4) 配置为其他值：说明这是一个配置参数，主要在makefile文件中利用。

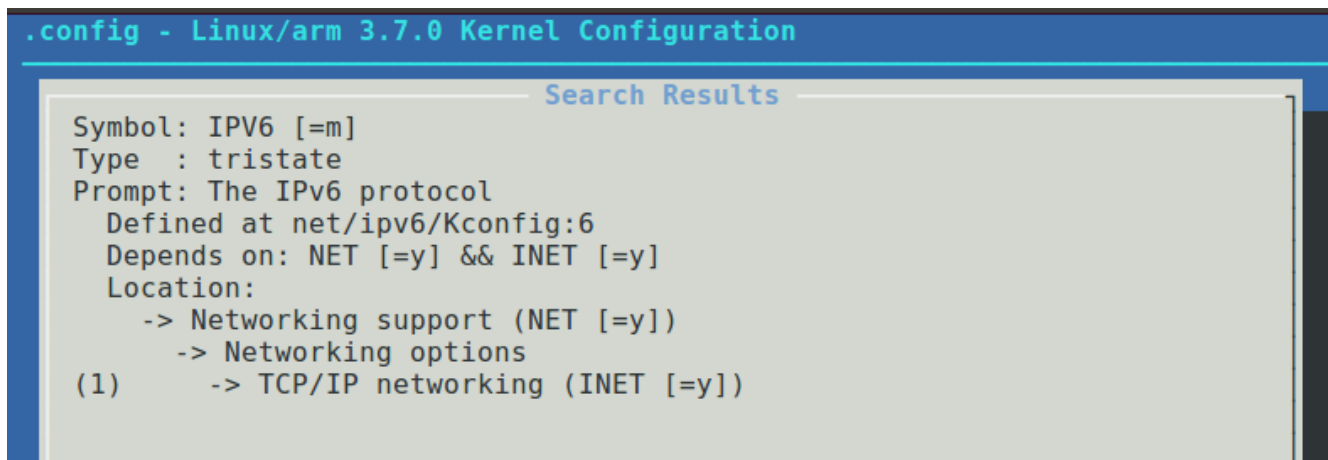
3、图形界面配置编辑器menuconfig配置过程

图形界面配置编辑器其实不只有menuconfig，还有其他的，看个人习惯选择合适的图形界面配置编辑器，menuconfig是在嵌入式中比较常用的图形界面配置编辑器，因此我也使用它来讲解配置过程。

在命令行中，cd到linux内核根目录，然后输入make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- LOADADDR=0x8000 menuconfig就可以启动menuconfig图形用户配置界面，如下图所示。



这是启动menuconfig后的根菜单，具体用法该界面顶部有很好的注释。这里举例说明，比如上文.config片段中的CONFIG_IPV6，可以在/后输入IPV6或者CONFIG_IPV6来查找，然后显示如下。



在Symbol栏目中看到对应于.config文件的配置名和配置值为m，对应.config文件中的CONFIG_IPV6=m。Prompt显示了配置菜单中对应的菜单名，Location栏目中显示了配置该项的界面路径。我们通过路径找到该配置选项，发现它的值为M，然后改为*，保存退出后生成新的.config文件，发现CONFIG_IPV6=y，如我们所期望。

4、配置原理

前文讲述了如何配置一个linux内核和产生的.config，但是没有讲到配置的原理以及如何在菜单中定制配置选项。

```
menuconfig: $(obj)/mconf
$< $(Kconfig)
```

在scripts/kconfig/Makefile中有如上所示一段，当执行make menuconfig时会调用配置工具mconf，配置工具先读取arch/arm/Kconfig文件来生成配置界面，这个文件是所有配置文件的总入口，它会包含其他目录的Kconfig文件。Kconfig文件是配置界面的源文件，配置工具读取各个Kconfig文件生成配置界面给我们配置内核最后得到.config。在深入分析kconfig文件语法之前先讲一下配置菜单的架构和相关几个要素的关系与现象。

```
--- Networking support
  Networking options --->
[ ]  Amateur Radio support --->
< > CAN bus subsystem support --->
< > IrDA (infrared) subsystem support --->
< > Bluetooth subsystem support --->
< > RxRPC session sockets
-*- Wireless --->
```

```
-*- Transformation migrate database (EXPERIMENTAL)
[ ] Transformation statistics (EXPERIMENTAL)
<*> PF_KEY sockets
[*] PF_KEY MIGRATE (EXPERIMENTAL)
[*] TCP/IP networking
[*] IP: multicasting
[ ] IP: advanced router
[*] IP: kernel level autoconfiguration
[*] IP: DHCP support
```

```
[ ] TCP: advanced congestion control --->
[ ] TCP: MD5 Signature Option support (RFC2385) (EXPERIMENTAL)
<M> The IPv6 protocol --->
[ ] Security Marking
[ ] Timestamping in PHY devices
[ ] Network packet filtering framework (Netfilter) --->
```

以上三图中深蓝色的条目代表了配置界面典型的三种结构。第一种是最左边没有任何括号，最右边有--->，在kconfig文件中的条目为menu。第二种最左边有括号，最右边没有--->，在kconfig文件中的条目为config。第三种最左边有括号，最右边有--->，在kconfig文件中的条目为menuconfig。

kconfig语法与作用分析：

(1) source

Kconfig中有很多类似于 source "drivers/pci/Kconfig" 这样的指令，这条指令告诉配置工具从内核源码树的drivers/pci/位置读取Kconfig，每种架构都有很多这样的Kconfig文件，配置工具将所有的Kconfig文件组合起来成为一个完整的配置集合，当用户配置内核时，配置集合以菜单的形式展现在用户面前。

(2) config和menuconfig

config条目是一个选项的配置条目，比如下面代码：

```
config TMPFS_XATTR
bool "Tmpfs extended attributes"
depends on TMPFS
default n
help
  Extended attributes are name:value pairs associated with inodes by
  the kernel or by users (see the attr(5) manual page, or visit
  <http://acl.bestbits.at/> for details).

  Currently this enables support for the trusted.* and
  security.* namespaces.
```

You need this **for** POSIX ACL support on tmpfs.

If unsure, say N.

这是fs/kconfig文件中的一段关于TMPFS_XATTR的配置选项，最终用户会利用该配置选项来配置生成一个CONFIG_TMPFS_XATTR=XXX或者#CONFIG_TMPFS_XATTR XXX的形式的条目在.config文件中。该文件中的bool为变量类型，有y和n两种取值，“Tmpfs extended attributes”为菜单中对应选项的显示的字符串。depends on TMPFS表示当前的配置依赖于TMPFS，当TMPFS被选中时才会显示当前配置选项。default n表示默认配置值为不配置。help为帮助信息。

menuconfig与config基本类似，除了多出一个if语法，例如。

```
menuconfig IPV6
    tristate "The IPv6 protocol"
    default m
    ---help---
    This is complemental support for the IP version 6.
    You will still be able to do traditional IPv4 networking as well.

    For general information about IPv6, see
    <http://playground.sun.com/pub/ipng/html/ipng-main.html>.
    For Linux IPv6 development information, see <http://www.linux-ipv6.org>.
    For specific information about IPv6 under Linux, read the HOWTO at
    <http://www.bieringer.de/linux/IPv6/>.

    To compile this protocol support as a module, choose M here: the
    module will be called ipv6.

if IPV6

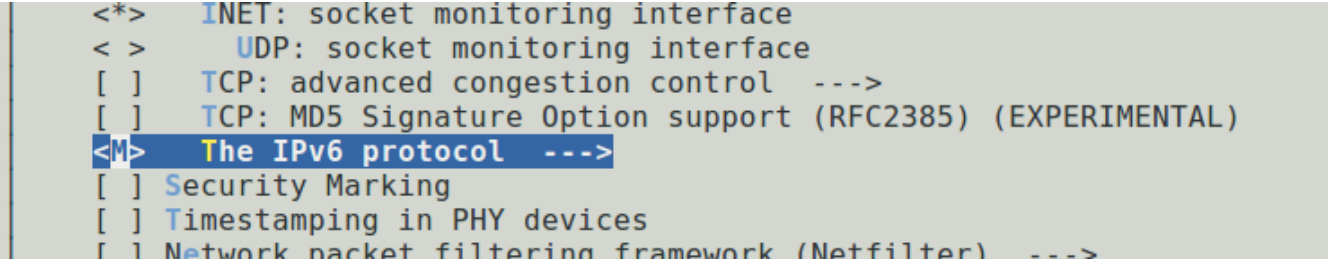
config IPV6_PRIVACY
    ...

config IPV6_ROUTER_PREF
    ...

...

endif # IPV6
```

这是net/ipv6/kconfig文件中关于IPV6的menuconfig配置，tristate和bool类似，但是它比bool多出一个m值。关键看最后的if endif的部分，意思是当IPV6被选中，即选M或者*时，可以看到接下来的其他配置项，否则看不到，如下图所示。



```
<*> INET: socket monitoring interface
< >  UDP: socket monitoring interface
[ ]   TCP: advanced congestion control  --->
[ ]   TCP: MD5 Signature Option support (RFC2385) (EXPERIMENTAL)
<M>  The IPv6 protocol  --->
[ ]   Security Marking
[ ]   Timestamping in PHY devices
[ ]   Network packet filtering framework (Netfilter)  --->
```

-- The IPv6 protocol

```
[ ] IPv6: Privacy Extensions (RFC 3041) support
[ ] IPv6: Router Preference (RFC 4191) support
[ ] IPv6: Enable RFC 4429 Optimistic DAD (EXPERIMENTAL)
< > IPv6: AH transformation
< > IPv6: ESP transformation
< > IPv6: IPComp transformation
< > IPv6: Mobility (EXPERIMENTAL)
<M> IPv6: IPsec transport mode
<M> IPv6: IPsec tunnel mode
<M> IPv6: IPsec BEET mode
< > IPv6: MIPv6 route optimization mode (EXPERIMENTAL)
<M> IPv6: IPv6-in-IPv4 tunnel (SIT driver)
[ ] IPv6: IPv6 Rapid Deployment (6RD) (EXPERIMENTAL)
< > IPv6: IP-in-IPv6 tunnel (RFC2473)
< > IPv6: GRE tunnel
[ ] IPv6: Multiple Routing Tables
[ ] IPv6: multicast routing (EXPERIMENTAL)
```

```
< > INET: socket monitoring interface
< > UDP: socket monitoring interface
[ ] TCP: advanced congestion control --->
[ ] TCP: MD5 Signature Option support (RFC2385) (EXPERIMENTAL)
< > The IPv6 protocol --->
[ ] Security Marking
[ ] Timestamping in PHY devices
```

-- The IPv6 protocol

(3) menu

menu的功能也和menuconfig类似，但不同的是menuconfig可以不选中来使得子配置项看不到，而menu的子配置项总是可见。示例代码arch/arm/kconfig一段如下所示，menu和endmenu之间的就是子配置项。

```
menu "System Type"

config MMU
...

choice
    prompt "ARM system type"
    default ARCH_MULTIPLATFORM

config ARCH_MULTIPLATFORM
...

...
```

```
endmenu
```

(4) choice

choice条目将多个配置项组合在一起供选择，但只能选择一条，如arch/arm/kconfig一段下图所示。

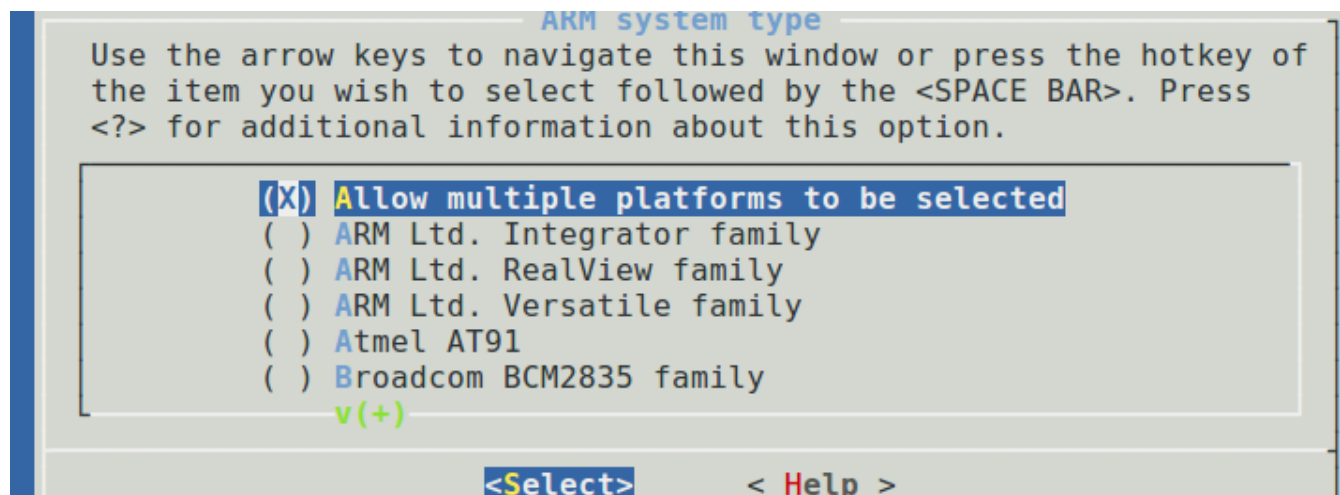
```
choice
    prompt "ARM system type"
    default ARCH_MULTIPLATFORM

config ARCH_MULTIPLATFORM
    bool "Allow multiple platforms to be selected"
    ...

config ARCH_INTEGRATOR
    bool "ARM Ltd. Integrator family"
    ...

...

endchoice
```



(5) comment

comment定义提示信息，出现在紧接着menu出现。

二、linux内核编译

未分析。

三、linux内核启动

1、内核镜像构造

在配置完后可以直接通过make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- LOADADDR=0x8000 ulmage来进行编译。得到下面最后的内核编译打印信息。

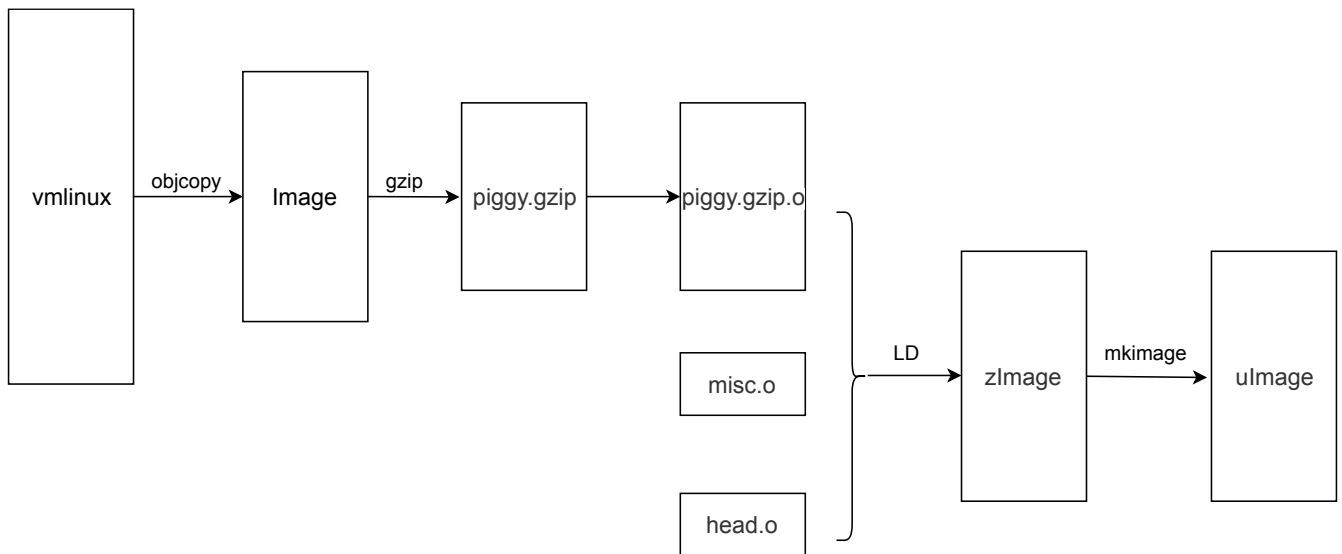
```
.....
LD      vmlinux
SYSMAP  System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy.gzip
AS      arch/arm/boot/compressed/piggy.gzip.o
CC      arch/arm/boot/compressed/misc.o
CC      arch/arm/boot/compressed/decompress.o
CC      arch/arm/boot/compressed/string.o
AS      arch/arm/boot/compressed/lib1funcs.o
AS      arch/arm/boot/compressed/ashldi3.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
UIMAGE arch/arm/boot/uImage
Image Name:   Linux-3.7.0
Created:      Fri Jun 12 09:12:41 2020
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    2703240 Bytes = 2639.88 kB = 2.58 MB
Load Address: 0x00008000
Entry Point:  0x00008000
Image arch/arm/boot/uImage is ready
```

以上信息描述了内核构建的几个主要对象，内核镜像主要有vmlinux、Image、piggy.gzip、zImage、ulmage。

- (1) vmlinux是elf格式的内核主体镜像；
- (2) Image是利用objcopy工具去除vmlinux中不必要段的段比如调试信息和符号表后得到的二进制格式文件；
- (3) piggy.gzip是利用gzip工具将Image压缩得到的压缩镜像，注意下面的piggy.gzip.S文件，该文件被汇编成目标文件piggy.gzip.o，它的作用是将压缩镜像piggy.gzip放进piggy.gzip.o的.piggydata段中，然后它可以和misc.o链接在一起，misc.o中有解压缩piggy.gzip的函数。

```
.section .piggydata,#alloc
.globl input_data
input_data:
    .incbin "arch/arm/boot/compressed/piggy.gzip"
    .globl input_data_end
input_data_end:
```

- (4) zImage是最终的合成的内核镜像，是通过将Image压缩后再额外在头部链接head.o和misc.o后得到的；
- (5) ulmage是利用mkimage工具在zImage镜像头部加了一个64字节的头部得到的，这个头部包含内核相关信息比如入口地址，uboot通常使用ulmage镜像启动内核。



上图展示了从vmlinux得到其他镜像的流程，下面重点分析u-boot启动ulImage的过程。uboot通过bootm ulImage_addr - fdt_addr命令启动，ulImage_addr是uboot将ulImage加载到内存的地址，在我们的socfpga工程中是0x7fc0，这个地址就是64字节头部的起始地址， $0x7fc0 + 0x40 (64) = 0x8000$ ，刚好是编译ulImage传入的参数，即入口地址（zImage第一条代码地址）。uboot从0x7fc0提取这64字节头部信息，验证ok后，设置好传给内核的参数（机器id和设备树地址）后跳到zImage入口地址。

zImage入口在head.o中，需要注意的是Image中也有head.o，Image才是真正的内核主体。我们将zImage头部的head.o和misc.o组合称为启动加载程序(bootstrap loader)，它是uboot和内核之间的过渡，主要负责提供合适的上下文给内核运行和执行必要的解压缩和重新部署内核二进制镜像。其中：

head.o负责：底层的、用汇编实现的处理器初始化，包括处理器的内部指令缓存、数据缓存、禁止中断和建立c语言环境。

misc.o负责：解压缩和部署内核。

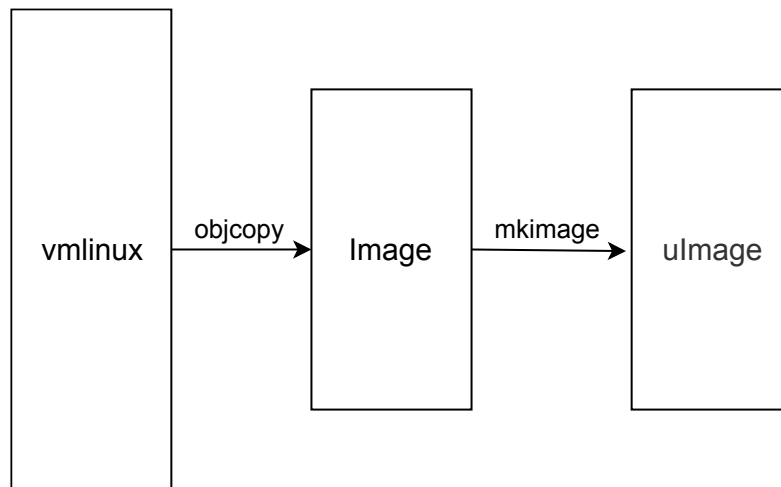
需要注意的和稍微修改的是以上是压缩版的内核启动方式，而本工程是用的非压缩版的内核启动，ulImage是直接通过Image加上64字节头部得到的。如下图启动信息的Image Type条目所示：

```

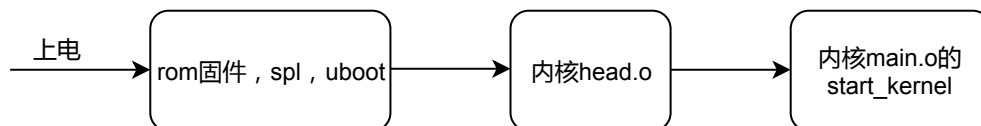
---
## Booting kernel from Legacy Image at 00007fc0 ...
   Image Name:   Linux-3.7.0
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2703240 Bytes = 2.6 MiB
   Load Address: 00008000
   Entry Point:  00008000
## Flattened Device Tree blob at 00000100
   Booting using the fdt blob at 0x00000100
   XIP Kernel Image ... OK
OK
   Loading Device Tree to 0fffa000, end 0ffffef3e ... OK

starting kernel ...
  
```

因此本工程的ulImage制造流程如下：



2、内核启动流程



2.1 uboot到内核入口head.o

在U-BOOT启动过程一文中分析到uboot最后启动内核阶段在boot_jump_linux中，注意在通过kernel_entry(0, machid, r2)真正跳转到内核之前调用了announce_and_cleanup()，其中printf("\nStarting kernel ...\n\n")就是上文中的打印信息Starting kernel ...，之后的打印信息就是内核启动自己输出的打印信息了。

```
static void boot_jump_linux(bootm_headers_t *images)
{
    ...

    announce_and_cleanup();

#ifdef CONFIG_OF_LIBFDT
    if (images->ft_len)
        r2 = (unsigned long)images->ft_addr;
    else
#endif
        r2 = gd->bd->bi_boot_params;

    kernel_entry(0, machid, r2);
}
```

```
static void announce_and_cleanup(void)
{
    printf("\nStarting kernel ...\n\n");

    ...
}
```

2.2 内核入口：head.o

内核入口模块head.o是由arch/arm/kernel/head.S生成的。下面展示了head.o模块的主分支代码。

```
//arch/arm/kernel/head.S
ENTRY(stext)

...

mrc p15, 0, r9, c0, c0      @ get processor id
bl  __lookup_processor_type  @ r5=procinfo r9=cpuid
movs    r10, r5             @ invalid processor (r5=0)?

beq __error_p               @ yes, error 'p'

...

ldr r8, =PHYS_OFFSET        @ always constant in this case

/*
 * r1 = machine no, r2 = atags or dtb,
 * r8 = phys_offset, r9 = cpuid, r10 = procinfo
 */
bl  __vet_atags              @ 验证atags或者dtb的有效性

...

bl  __create_page_tables

ldr r13, =__mmap_switched    @ 当mmu被使能后跳转的地址，同时设定栈指针

...

1:  b  __enable_mmu
ENDPROC(stext)

...

__enable_mmu:

...

b  __turn_mmu_on
ENDPROC(__enable_mmu)

ENTRY(__turn_mmu_on)
mov r0, r0
instr_sync
```

```

    mcr p15, 0, r0, c1, c0, 0      @ write control reg
    mrc p15, 0, r3, c0, c0, 0      @ read id reg
    instr_sync
    mov r3, r3
    mov r3, r13
    mov pc, r3                    @ 跳到__mmap_switched
__turn_mmu_on_end:
ENDPROC(__turn_mmu_on)

...

#include "head-common.S"

```

```

//arm/kernel/head-common.S

...

__mmap_switched:
    ...

    cmp r4, r5                    @ Copy data segment if needed
1:  cmpne r5, r6
    ldrne fp, [r4], #4
    strne fp, [r5], #4
    bne 1b

    mov fp, #0                    @ Clear BSS (and zero fp)
1:  cmp r6, r7
    strcc fp, [r6], #4
    bcc 1b
    ...

    str r9, [r4]                  @ Save processor ID
    str r1, [r5]                  @ Save machine type
    str r2, [r6]                  @ Save atags pointer
    bic r4, r0, #CR_A             @ Clear 'A' bit
    stmia r7, {r0, r4}           @ Save control register values
    b    start_kernel
ENDPROC(__mmap_switched)

...

```

由以上代码可以总结出head.o模块主要工作：

- (1) 检测处理器的有效性。
- (2) 创建初始的页表，这只是一个最基本的页表，满足当前阶段内核的运行，通常只是仅映射内核代码本身，最后会在start_kernel阶段的初始化中重新映射。
- (3) 启动mmu。
- (4) 设置好c语言环境。
- (5) 跳转到start_kernel中初始化内核。

2.3 内核初始化：start_kernel

start_kernel是真正的内核初始化函数，大部分内核初始化工作在这里完成，由于涉及较多专业的知识且比较庞大，这里提取出开发人员主要关注的地方。

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;

    printk(KERN_NOTICE "%s", linux_banner);
    setup_arch(&command_line);

    ...

    setup_command_line(command_line);

    ...

    printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
    parse_early_param();
    parse_args("Booting kernel", static_command_line, __start__param,
               __stop__param - __start__param,
               -1, -1, &unknown_bootoption);
    ...

    console_init();

    ...

    rest_init();
}
```

```
Linux version 3.7.0 (root@tg) (gcc version 4.7.3 20121106 (prerelease) (crosstool-NG linaro-1.13.1-4.7-2012.11-20121123 - Linaro GCC
2012.11)) #1 SMP PREEMPT Tue Nov 26 09:43:46 CST 2019
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine: Altera SOC FPGA, model: Altera SOC FPGA Cyclone V
reserve 32MB@0x1E000000 for Bare Metal
Memory policy: ECC disabled, Data cache writealloc
PERCPU: Embedded 8 pages/cpu @80d4c000 s10944 r8192 d13632 u32768
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 251904
Kernel command line: console=ttyS0,57600 root=/dev/mmcblk0p2 rw rootwait
PID hash table entries: 4096 (order: 2, 16384 bytes)
```

注意上图第一句打印信息就是内核第一条打印输出，输出内核版本对应代码中的printk(KERN_NOTICE "%s", linux_banner)，第二条标红的是内核打印的命令行信息，对应于代码中的printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line)。开发人员其实关注最多的就是setup_arch、setup_command_line和rest_init，下面——分析他们。

2.3.1 setup_arch(&command_line)

```
void __init setup_arch(char **cmdline_p)
{
    struct machine_desc *mdesc;

    setup_processor();
    mdesc = setup_machine_fdt(__atags_pointer);
    ...
    machine_desc = mdesc;
    machine_name = mdesc->name;
```

```

...

strcpy(cmd_line, boot_command_line, COMMAND_LINE_SIZE);
*cmdline_p = cmd_line;
parse_early_param();

...

paging_init(mdesc);

...

unflatten_device_tree();

...

}

```

setup_arch函数先通过**setup_processor()**函数获取处理器信息，并设置处理器和将处理器信息打印出来。然后通过**setup_machine_fdt**获取板子的机器描述符，并且从设备树中提取命令行参数到boot_command_line中，如下所示。注意该命令行参数是经修改过后的设备树，在uboot最后启动时修改得到的最终的设备树。

```

struct machine_desc * __init setup_machine_fdt(unsigned int dt_phys)
{
    ...

    // 获取板子的机器描述符
    for_each_machine_desc(mdesc) {
        score = of_flat_dt_match(dt_root, mdesc->dt_compat);
        if (score > 0 && score < mdesc_score) {
            mdesc_best = mdesc;
            mdesc_score = score;
        }
    }

    ...

    // 从设备树中提取命令行参数到boot_command_line中
    of_scan_flat_dt(early_init_dt_scan_chosen, boot_command_line);

    ...
}

```

```

//arch/arm/mach-socfpga/socfpga.c
static const char *altera_dt_match[] = {
    "altr,socfpga",
    "altr,socfpga-cyclone5",
    "altr,socfpga-vt",
    "altr,socfpga-ice",
    NULL
}

```

```
};

DT_MACHINE_START(SOCFPGA, "Altera SOCFPGA")
    .smp            = smp_ops(socfpga_smp_ops),
    .map_io         = socfpga_map_io,
    .init_irq       = gic_init_irq,
    .handle_irq     = gic_handle_irq,
    .timer          = &dw_apb_timer,
    .nr_irqs        = SOCFPGA_NR_IRQS,
    .init_machine   = socfpga_cyclone5_init,
    .restart        = socfpga_cyclone5_restart,
    .reserve        = socfpga_ucosii_reserve,
    .dt_compat      = altera_dt_match,
MACHINE_END
```

```
//arch/arm/include/asm/mach/arch.h
#define DT_MACHINE_START(_name, _namestr) \
static const struct machine_desc __mach_desc_##_name \
__used \
__attribute__((__section__(".arch.info.init"))) = { \
    .nr            = ~0, \
    .name          = _namestr,
```

机器描述符是通过DT_MACHINE_START和MACHINE_END宏生成的一个struct machine_desc结构体，链接时放在一个.arch.info.init段中，该段放很多内核支持的机器描述符。

在获取了机器描述符后调用**parse_early_param()**来处理早期的命令行参数，在讲处理命令行之前先讲一下如何将某个命令行参数与该命令行参数处理函数绑定的机制。

```
//include/linux/init.h
struct obs_kernel_param {
    const char *str;
    int (*setup_func)(char *);
    int early;
};

#define __setup_param(str, unique_id, fn, early) \
    static const char __setup_str_##unique_id[] __initconst \
    __aligned(1) = str; \
    static struct obs_kernel_param __setup_##unique_id \
    __used __section(.init.setup) \
    __attribute__((aligned((sizeof(long))))) \
    = { __setup_str_##unique_id, fn, early }

#define __setup(str, fn) \
    __setup_param(str, fn, fn, 0)

#define early_param(str, fn) \
    __setup_param(str, fn, fn, 1)
```

通过setup或者early_param宏注册某个命令行处理函数，最后会调用 setup_param宏来生成一个 obs_kernel_param结构体，将命令行参数名(比如console=ttyS0,115200中的console)，处理函数指针还有一个 early标志位填入该结构体，该结构体会链接到一个.init.setup的特殊段中，之后处理命令行就从这个段中搜索匹配的结构体。特别要说明的是early标记，setup宏定义的early标记为0，说明这不是早期处理的命令行参数，early_param宏定义的early标记为1，说明这是早期处理的参数，会在parse_early_param()中的do_early_param函数中处理，至于会不会在以后的非早期处理函数中重新处理，看自己的情况，一般不会，简而言之就是setup宏定义的在parse_early_param()之外处理，该文中是unknown_bootoption中的obsolete_checksetup，early_param宏定义的在parse_early_param中的do_early_param里处理。

```
void __init parse_early_options(char *cmdline)
{
    parse_args("early options", cmdline, NULL, 0, 0, 0, do_early_param);
}

/* Arch code calls this early on, or if not, just before other parsing. */
void __init parse_early_param(void)
{
    static __initdata int done = 0;
    static __initdata char tmp_cmdline[COMMAND_LINE_SIZE];

    if (done)
        return;

    /* All fall through to do_early_param. */
    strcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
    parse_early_options(tmp_cmdline);
    done = 1;
}
```

```
/* Chew leading spaces */
int parse_args(const char *doing,
               char *args,
               const struct kernel_param *params,
               unsigned num,
               s16 min_level,
               s16 max_level,
               int (*unknown)(char *param, char *val, const char *doing))
{
    ...

    while (*args) {
        ...
        args = next_arg(args, &param, &val);
        ...
        ret = parse_one(param, val, doing, params, num,
                        min_level, max_level, unknown);
        ...
    }

    /* All parsed OK. */
    return 0;
}
```



```
}
```

```
static int __init do_early_param(char *param, char *val, const char *unused)
{
    const struct obs_kernel_param *p;

    for (p = __setup_start; p < __setup_end; p++) {
        if ((p->early && parameq(param, p->str)) ||
            (strcmp(param, "console") == 0 &&
             strcmp(p->str, "earlycon") == 0)
        ) {
            if (p->setup_func(val) != 0)
                printk(KERN_WARNING
                       "Malformed early option '%s'\n", param);
        }
    }
    /* We accept everything at this stage. */
    return 0;
}
```

上面几个函数清除说明了parse_early_param是如何处理早期的命令行参数的。parse_early_param首先调用parse_early_options(tmp_cmdline)，然后调用parse_args。parse_args中的while循环用来处理完所有的命令行参数，每次循环处理一个，在一次循环中，通过next_arg提取一个命令行名和值，然后用parse_one处理该命令行参数，传给unknown的就是do_early_param。do_early_param从之前存放定义的obs_kernel_param结构的段中寻找early标记不为0且参数名匹配的obs_kernel_param结构体，然后调用该结构体中的处理函数。

2.3.2 setup_command_line(command_line)

```
static void __init setup_command_line(char *command_line)
{
    saved_command_line = alloc_bootmem(strlen (boot_command_line)+1);
    static_command_line = alloc_bootmem(strlen (command_line)+1);
    strcpy (saved_command_line, boot_command_line);
    strcpy (static_command_line, command_line);
}
```

该函数只是设置了saved_command_line和static_command_line两个全局变量为内核命令行。

2.3.3 parse_early_param()

在setup_arch(&command_line)中已经调用parse_early_param()处理了早期命令行，个人觉得这里是多余的。

2.3.4 parse_args(..., &unknown_bootoption)

该函数的流程在2.3.1中讲到，这次不同的是传入的函数是unknown_bootoption不是do_early_param，该函数就是用来处理早期命令行参数之外的。

```
static int __init unknown_bootoption(char *param, char *val, const char *unused)
{
    ...

    /* Handle obsolete-style parameters */
    if (obsolete_checksetup(param))
        return 0;

    ...
}
```

该函数中重点是obsolete_checksetup，该函数才是核心。

```
static int __init obsolete_checksetup(char *line)
{
    const struct obs_kernel_param *p;
    int had_early_param = 0;

    p = __setup_start;
    do {
        int n = strlen(p->str);
        if (parameqn(line, p->str, n)) {
            if (p->early) {
                /* Already done in parse_early_param?
                 * (Needs exact match on param part).
                 * Keep iterating, as we can have early
                 * params and __setups of same names 8( */
                if (line[n] == '\0' || line[n] == '=')
                    had_early_param = 1;
            } else if (!p->setup_func) {
                printk(KERN_WARNING "Parameter %s is obsolete,"
                       " ignored\n", p->str);
                return 1;
            } else if (p->setup_func(line + n))
                return 1;
        }
        p++;
    } while (p < __setup_end);

    return had_early_param;
}
```

该函数只是将之前没有在parse_early_param()函数中处理的early标记为0的命令行参数处理。

```
static int __init root_dev_setup(char *line)
{
    strncpy(saved_root_name, line, sizeof(saved_root_name));
    return 1;
}
__setup("root=", root_dev_setup);

static int __init rootwait_setup(char *str)
```

```

{
    if (*str)
        return 0;
    root_wait = 1;
    return 1;
}
__setup("rootwait", rootwait_setup);

static int __init readwrite(char *str)
{
    if (*str)
        return 0;
    root_mountflags &= ~MS_RDONLY;
    return 1;
}
__setup("rw", readwrite);

```

以上是命令行参数中剩下部分的设置，即root，rw和rootwait，这几个参数都是文件系统相关的，帮助正确的挂载根文件系统。剩下的console参数在下一节的控制台初始化中分析。

2.3.5 console_init()

在console_init()之前的代码中有printk，但这时候只是将打印信息放到缓存区中，并没有输出到控制台(串口)。在console_init()进行控制台初始化前需要依赖一点前文的obsolete_checksetup函数中对console命令行参数的处理。这个处理在kernel/printk.c中。

```

static int __init console_setup(char *str)
{
    char buf[sizeof(console_cmdline[0].name) + 4]; /* 4 for index */
    char *s, *options, *brl_options = NULL;
    int idx;

    /*
     * Decode str into name, index, options.
     */
    if (str[0] >= '0' && str[0] <= '9') {
        strcpy(buf, "ttyS");
        strncpy(buf + 4, str, sizeof(buf) - 5);
    } else {
        strncpy(buf, str, sizeof(buf) - 1);
    }
    buf[sizeof(buf) - 1] = 0;
    if ((options = strchr(str, ',')) != NULL)
        *(options++) = 0;

    for (s = buf; *s; s++)
        if ((*s >= '0' && *s <= '9') || *s == ',')
            break;
    idx = simple_strtoul(s, NULL, 10);
    *s = 0;

    __add_preferred_console(buf, idx, options, brl_options);
}

```

```

    console_set_on_cmdline = 1;
    return 1;
}

__setup("console=", console_setup);

```

```

static int __add_preferred_console(char *name, int idx, char *options,
                                   char *brl_options)
{
    struct console_cmdline *c;
    int i;

    /*
     * See if this tty is not yet registered, and
     * if we have a slot free.
     */
    for (i = 0; i < MAX_CMDLINECONSOLES && console_cmdline[i].name[0]; i++)
        if (strcmp(console_cmdline[i].name, name) == 0 &&
            console_cmdline[i].index == idx) {
            if (!brl_options)
                selected_console = i;
            return 0;
        }
    if (i == MAX_CMDLINECONSOLES)
        return -E2BIG;
    if (!brl_options)
        selected_console = i;
    c = &console_cmdline[i];
    strncpy(c->name, name, sizeof(c->name));
    c->options = options;
#ifdef CONFIG_A11Y_BRAILLE_CONSOLE
    c->brl_options = brl_options;
#endif
    c->index = idx;
    return 0;
}

```

console_setup(char *str)中传入的参数是完整的console=ttyS0,57600，首先从这个字符串中提取信息，然后根据信息选取console_cmdline数组对应的一项，表示选中哪个串口，然后填充信息进去为后文的console_init()作准备。

```

//drivers/tty/tty_io.c
void __init console_init(void)
{
    initcall_t *call;

    /* Setup the default TTY line discipline. */
    tty_ldisc_begin();

    /*
     * set up the console device so that later boot sequences can
     * inform about problems etc..
     */
}

```

```

    */
    call = __con_initcall_start;
    while (call < __con_initcall_end) {
        (*call)();
        call++;
    }
}

```

```

//include/linux/init.h
#define console_initcall(fn) \
    static initcall_t __initcall_##fn \
    __used __section(.con_initcall.init) = fn

```

上述代码也是用了前文一样的技术，先通过console_initcall宏将控制台初始化函数放入.con_initcall.init段中，等到console_init初始化时就从该段中提取出来执行。

2.3.6 rest_init()

这是内核初始化最后步骤。主要做的是启动第一个用户空间程序。

```

static noinline void __init_refok rest_init(void)
{
    ...
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    ...
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
    ...
    cpu_idle();
}

```

从代码中可以看出rest_init()主要创建了一个kernel_init线程和kthreadd线程，包括start_kernel代表的线程总共3个线程。start_kernel线程在完成所有初始化后进入cpu_idle变成空闲进程，kthreadd线程是一个提供服务的后台进程，这里不需要过多关注。重点要关注的是kernel_init，它也称为init进程，是所有用户空间进程的父进程，进程id为1。

```

static int __ref kernel_init(void *unused)
{
    kernel_init_freeable();

    ...

    if (execute_command) {
        if (!run_init_process(execute_command))
            return 0;
        printk(KERN_WARNING "Failed to execute %s. Attempting "
            "defaults...\n", execute_command);
    }
    if (!run_init_process("/sbin/init") ||
        !run_init_process("/etc/init") ||
        !run_init_process("/bin/init") ||
        !run_init_process("/bin/sh"))
        return 0;
}

```

```

panic("No init found. Try passing init= option to kernel. "
      "See Linux Documentation/init.txt for guidance.");
}

```

```

static void __init kernel_init_freeable(void)
{
    ...

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

    ...
    prepare_namespace();
    ...
}

void __init prepare_namespace(void)
{
    ...
    mount_root();
    ...
}

static int __init init_setup(char *str)
{
    ...

    execute_command = str;
    ...
}

__setup("init=", init_setup);

```

在运行第一个用户空间程序之前先要打开控制台和挂载根文件系统，挂载根文件系统路径为 kernel_init_freeable() -> prepare_namespace() -> mount_root()。在挂载了根文件系统后通过检查 execute_command 是否在命令行传入自定义用户空间程序，命令行形式为 init=XXX，本工程并没有设置该参数，因此 execute_command 为空，所以通过 run_init_process 按 /sbin/init、/etc/init、/bin/init、/bin/sh 的顺序直到有一个成功，调用 execve 将新程序覆盖当前进程，然后不再返回，如果一个都不成功则系统打印出错信息并且启动失败。

注意：这里打开控制台出现在挂载根文件系统之前，而 /dev/console 就是根文件系统的内容，这明显矛盾，原因是在挂载真正的文件系统之前会有个虚拟的根文件系统，在 vfs_caches_init 函数中实现，具体实现由于目前水平和时间的原因暂不分析，只说它大概的作用，它在内存中而不在闪存中，因为此时内核还没有加载闪存相关驱动程序，它是一个过渡阶段，相当于一个早期的根文件系统。

2.4 用户空间初始化

如果没有在命令行指定 init 程序，那通常第一个用户空间程序就是 /sbin/init，本工程的根文件系统的 sbin 目录中就有 init，无论是哪个我们都用 init 来表示。**init 的主要功能是根据一个特定的配置文件生成其他进程。**这个配置文件在根文件系统的 /etc/inittab。而这个 init 程序是和根文件系统一起构建的，来源于某些构建系统，比如 busybox，buildroot，yocto 等，本工程的根文件系统就是用 yocto 构建的。其中 buildroot 和 yocto 不单只能构建根文件系统，还能构建内核镜像和 uboot 镜像，而 busybox 只能构建根文件系统和相关 linux 命令行工具，但是 busybox 裁减性很

高，能做出非常精简的文件系统，在下一文中具体讲解如何使用busybox构建一个根文件系统替换本工程的根文件系统。

接下来讲的是在假设通过busybox制作的根文件系统并挂载上了之后，init程序的启动步骤。busybox的init和标准的System V init启动(比如yocto)稍有区别，busybox的inittab不使用运行级别。

/etc/inittab文件中每个条目用来定义一个子进程，并确定它们的启动方法，格式如下：

...

各个字段的作用如下：

- : 表示这个子进程使用的控制台，如果省略则使用和init一样的控制台。
- : busybox不使用。
- : 表示init进程如何控制该子进程。

action	执行条件	说明
sysinit	系统启动后最先执行	只执行一次，init进程等待它结束才执行其他动作
wait	系统执行完sysinit进程后	只执行一次，init进程等待它结束才执行其他动作
once	系统执行完wait进程后	只执行一次，init进程不等待它结束
respawn	系统启动完once进程后	init进程监测发现该子进程退出时，重启它
askfirst	系统启动完respawn进程后	与respawn类似，不过init进程先输出“Please press Enter to activate this console.”等待用户输入回车键之后才启动该子进程。
shutdown	系统关机时	重启、关闭系统时
restart	busybox中配置了CONFIG_FEATURE_USE_INITTAB，并且init进程接收到SIGHUP信号时	先重新读取、解析/etc/inittab文件，再执行restart程序

: 要执行的程序，可以使可执行程序也可以是执行脚本，如果前有-，则是交互程序。

再来看看利用busybox新制作的/etc/inittab测试文件：

```
# yeshen inittab

::sysinit:/etc/init.d/rcs

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

/etc/init.d/rcS文件:

```
#!/bin/sh

echo "this is first rcs"
```



```
waiting for root device /dev/mmcblk0p2...
mmc_host mmc0: Bus speed (slot 0) = 125000000Hz (slot req 125000000Hz, actual 125000000HZ div = 0)
mmc0: new high speed SDHC card at address 1234
mmcblk0: mmc0:1234 SA32G 28.8 GiB
mmcblk0: p1 p2 p3
kjournald starting. Commit interval 5 seconds
EXT3-fs (mmcblk0p2): using internal journal
EXT3-fs (mmcblk0p2): mounted filesystem with ordered data mode
VFS: Mounted root (ext3 filesystem) on device 179:2.
devtmpfs: mounted
Freeing init memory: 148K
this is first rcs
```

Please press Enter to activate this console. ++OTG Interrupt: A-Device Timeout Change++

启动后信息如上，系统启动后的行为和分析一致。