

06-2 Adding to the Kernel, Kernel Initialization

Adding to the Kernel

- Makefile Targets
- Kernel Configuration
- Custom Configuration Options
- Kernel Makefiles
- Kernel Documentation

Composite Kernel Image

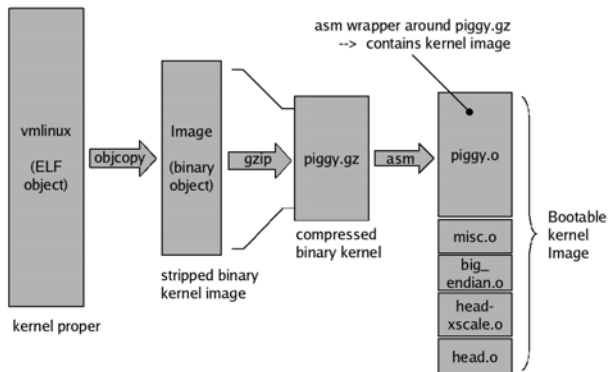


Figure 5.1 page 103

piggy.S

```

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

```

Bootstrap Loader (not bootloader)

- Prove context for kernel
 - Enable instruction set
 - Data caches
 - Disable interrupt
 - C runtime environment
- Decompress (misc.o)
- Relocate kernel image

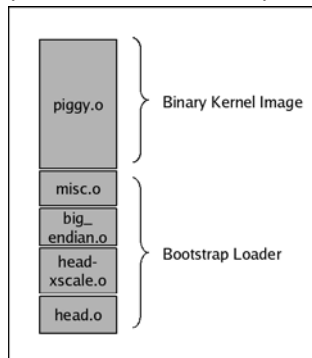


Figure 5-2 page 105

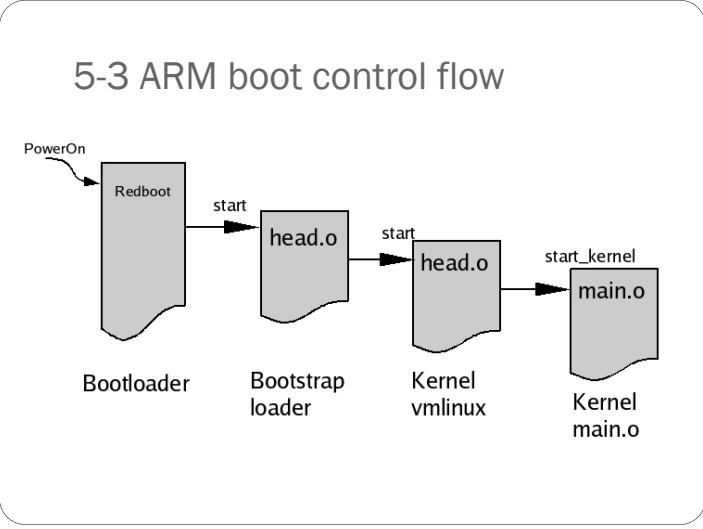
Boot Messages

- See handout
- Note *kernel version string*
- Note *kernel command line*

5-3 ARM boot control flow

```
graph LR; PowerOn --> Redboot; Redboot -- start --> head1[head.o]; head1 -- start --> head2[head.o]; head2 -- start_kernel --> main[main.o];
```

The diagram illustrates the ARM boot control flow. It begins with a 'PowerOn' event, indicated by a wavy arrow, which triggers the execution of the 'Redboot' bootloader. The 'Redboot' stage is represented by a grey box with a wavy bottom edge. An arrow labeled 'start' points from 'Redboot' to the first 'head.o' block, which is also a grey box with a wavy bottom edge. This 'head.o' block is labeled 'Bootstrap loader' below it. Another arrow labeled 'start' points from this 'head.o' block to a second 'head.o' block, which is labeled 'Kernel vmlinux' below it. A final arrow labeled 'start_kernel' points from the second 'head.o' block to the 'main.o' block, which is labeled 'Kernel main.o' below it. The 'main.o' block is a grey box with a wavy bottom edge.



Booting Linux – ROM to Kernel

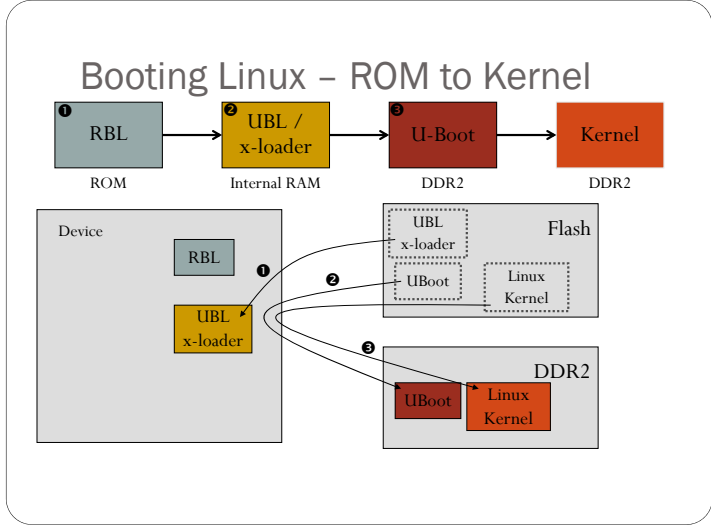
The diagram illustrates the booting process from ROM to Kernel. It consists of two parts: a high-level flowchart at the top and a detailed hardware-level diagram at the bottom.

High-level flowchart:

- 1 RBL** (ROM) → **2 UBL / x-loader** (Internal RAM) → **3 U-Boot** (DDR2) → **Kernel** (DDR2)

Detailed hardware-level diagram:

- Device:** Contains **RBL** and **UBL x-loader**.
 - 1:** Arrow from **RBL** to **UBL x-loader**.
 - 2:** Arrow from **UBL x-loader** to **UBL x-loader** in **Flash**.
 - 3:** Arrow from **UBL x-loader** to **UBoot** in **DDR2**.
- Flash:** Contains **UBL x-loader** and **Linux Kernel** (dashed boxes).
- DDR2:** Contains **UBoot** and **Linux Kernel** (solid boxes).
 - 4:** Arrow from **UBoot** to **Linux Kernel**.



```

.../arch/arm/boot/compressed/head.S

#include <linux/linkage.h>

#ifdef DEBUG

    #if defined(CONFIG_DEBUG_ICEDCC)

        .macro writeb, ch, rb
        senduart \ch, \rb
        .endm

    #endif CONFIG_CPU_V6

    .macro loadsp, rb
    .endm

    .macro writeb, ch, rb
    mcr pl4, 0, \ch, c0, c5, 0
    .endm

#else

    .macro loadsp, rb
    .endm

    .macro writeb, ch, rb
    mcr pl4, 0, \ch, c1, c0, 0
    .endm

#endif

#
#else

#include <mach/debug-macro.S>

        .macro writeb, ch, rb
        senduart \ch, \rb
        .endm

    #if defined(CONFIG_ARCH_SA1100)
        .macro loadsp, rb
        mov \rb, #0x80000000 @
        physical base address
    .endm
    #endif
#endif

```

```

.../arch/arm/boot/compressed/head.S

#include <linux/linkage.h>

#ifdef DEBUG

    #if defined(CONFIG_DEBUG_ICEDCC)

        .macro writeb, ch, rb
        senduart \ch, \rb
        .endm

    #endif CONFIG_CPU_V6

    .macro loadsp, rb
    .endm

    .macro writeb, ch, rb
    mcr pl4, 0, \ch, c0, c5, 0
    .endm

#else

    .macro loadsp, rb
    .endm

    .macro writeb, ch, rb
    mcr pl4, 0, \ch, c1, c0, 0
    .endm

#endif

#
#else

#include <mach/debug-macro.S>

        .macro writeb, ch, rb
        senduart \ch, \rb
        .endm

    #if defined(CONFIG_ARCH_SA1100)
        .macro loadsp, rb
        mov \rb, #0x80000000 @
        physical base address
    .endm
    #endif
#endif

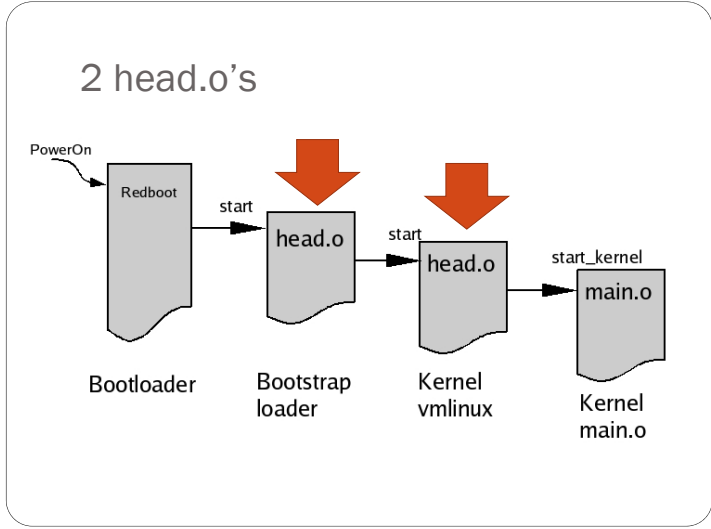
```

2 head.o's

```
graph LR; PowerOn --> Redboot[Redboot]; Redboot -- start --> head1[head.o]; head1 -- start --> head2[head.o]; head2 -- start_kernel --> main[main.o];
```

The diagram illustrates the boot process flow:

- PowerOn** (indicated by a wavy arrow) triggers the **Redboot** (Bootloader).
- Redboot** (Bootloader) calls **start** to load the first **head.o** (Bootstrap loader).
- The first **head.o** (Bootstrap loader) calls **start** to load the second **head.o** (Kernel vmlinux).
- The second **head.o** (Kernel vmlinux) calls **start_kernel** to load **main.o** (Kernel main.o).



.../arch/arm/kernel/head.S

1. Checks of valid processor and architecture
2. Creates initial page table entries
3. Enables the processor's memory management unit (MMU)
4. Establishes limited error detection and reporting
5. Jumps to the start of the kernel proper,
start_kernel() in **main.c**.

Find these on the handout

- ## .../arch/arm/kernel/head.S
1. Checks of valid processor and architecture
 2. Creates initial page table entries
 3. Enables the processor's memory management unit (MMU)
 4. Establishes limited error detection and reporting
 5. Jumps to the start of the kernel proper,
start_kernel() in **main.c**.
- Find these on the handout

.../arch/arm/kernel/head.S


1. Checks of valid processor and architecture
2. Creates initial page table entries
3. Enables the processor's memory management unit (MMU)
4. Establishes limited error detection and reporting
5. Jumps to the start of the kernel proper,
start_kernel() in **main.c**.

Find these on the handout


Kernel Startup

- `arch/arm/kernel/head.S`

`b start_kernel`




Find this by tomorrow

- # Kernel Startup
- `arch/arm/kernel/head.S`
- `b start_kernel`
- 
- Find this by tomorrow

Kernel Startup

- `arch/arm/kernel/head.S`

`b start_kernel`




Find this by tomorrow

Kernel Startup

- `arch/arm/kernel/head.S`

`b start_kernel`



Find this by tomorrow

.../init/main.c

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;
    extern struct kernel_param __start__param[], __stop__param[];

    smp_setup_processor_id();

    /*
     * Need to run as early as possible, to initialize the
     * lockdep hash:
     */
    lockdep_init();
    debug_objects_early_init();
    cgroup_init_early();

    local_irq_disable();
    early_boot_irqs_off();
    early_init_irq_lock_class();
}
```

Kernel Command Line Processing

- Read 5.3 on Kernel Command-Line Processing
- It presents the `__setup` macro

```
console=ttyS2,115200n8 mem=80M@0x80000000
mem=384M@0x88000000 mpu_rate=1000
buddy=none camera=lbc3m1 vram=16M
omapfb.vram=0:8M,1:4M,2:4M
omapfb.mode=dvi:hd720
omapdss.def_disp=dvi root=/dev/mmcblk0p2
rw rootfstype=ext3 rootwait
```

Console Setup Code Snippet

```
/*
 * Setup a list of consoles. Called from init/main.c
 */
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;
    ...
    <body omitted for clarity...>
    ...
    return 1;
}
__setup("console=", console_setup);
```

.../include/linux/init.h

```
/*
 * Only for really core code. See moduleparam.h for the normal way.
 */
/* Force the alignment so the compiler doesn't space elements of the
 * obs_kernel_param "array" too far apart in .init.setup.
 */
#define __setup_param(str, unique_id, fn, early) \
    static char __setup_str_##unique_id[] __initdata __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)
```

__setup

```
__setup("console=", console_setup);
```

- Expands to

```
static const char __setup_str_console_setup[] __initconst \
__aligned(1) = "console=";
static struct obs_kernel_param __setup_console_setup __used \
__section(.init.setup) __attribute__ \
((aligned((sizeof(long))))) \
= { __setup_str_console_setup, console_setup, early};
```

- Which expands to

```
static struct obs_kernel_param __setup_console_setup \
__section(.init.setup) = { __setup_str_console_setup, \
console_setup, early};
```

- This stores the code in a table in section `.init.setup`.

On initialization...

- The table in `.init.setup` has
 - Parameter string ("`console=`") and
 - Pointer to the function that processes it.
- This way the initialization code can process everything on the command line without knowing at compile time where all the code is.
- See section 5.3 for more details.