ARM64 U-BOOT 启动过程分析

Base on TI K3 platform

# David Zhou

**U-boot SPL阶段：**

在.config里面可以看到CONFIG\_SPL=y，我们的K3需要SPL支持。K3的镜像存储示意图：

+-----------------------+

| FIT HEADER |

| +-------------------+ |

| | A53 ATF | |

| +-------------------+ |

| | A53 OPTEE | |

| +-------------------+ |

| | A53 SPL | |

| +-------------------+ |

| | SPL DTB 1...N | |

| +-------------------+ |

+-----------------------+

可以看到SPL使用的Device Tree放在SPL镜像的后面。

为了搞清楚u-boot-spl的启动，我们首先来看一下u-boot的链接文件arch/arm/cpu/armv8/u-boot-spl.lds

MEMORY { .sram : ORIGIN = IMAGE\_TEXT\_BASE,

LENGTH = IMAGE\_MAX\_SIZE }

MEMORY { .sdram : ORIGIN = CONFIG\_SPL\_BSS\_START\_ADDR,

LENGTH = CONFIG\_SPL\_BSS\_MAX\_SIZE }

lds文件首先描述了两块内存空间。

第一块是在.sram里面，起始地址是IMAGE\_TEXT\_BASE，这个值定义在Makefile.spl下面。如果激活了CONFIG\_SPL，在u-boot顶层Makefile里面：

spl/u-boot-spl: tools prepare \

$(if $(CONFIG\_OF\_SEPARATE)$(CONFIG\_OF\_EMBED)$(CONFIG\_SPL\_OF\_PLATDATA),dts/dt.dtb) \

$(if $(CONFIG\_OF\_SEPARATE)$(CONFIG\_OF\_EMBED)$(CONFIG\_TPL\_OF\_PLATDATA),dts/dt.dtb)

$(Q)$(MAKE) obj=spl -f $(srctree)/scripts/Makefile.spl all

可以看到就是编译spl所需要使用Makefile.spl。这个文件位于u-boot/scripts下面：

KBUILD\_CPPFLAGS += -DCONFIG\_SPL\_BUILD

我们在文件里面经常会看到CONFIG\_SPL\_BUILD这个宏，这个宏就是在Makefile.spl定义的

ifneq ($(CONFIG\_$(SPL\_TPL\_)TEXT\_BASE),)

LDPPFLAGS += -DIMAGE\_TEXT\_BASE=$(CONFIG\_$(SPL\_TPL\_)TEXT\_BASE)

endif

CONFIG\_SPL\_TEXT\_BASE在.config文件里面已经定义：

CONFIG\_SPL\_TEXT\_BASE=0x80080000

IMAGE\_MAX\_SIZE 在Makefile.spl里面定义为：

ifneq ($(CONFIG\_$(SPL\_TPL\_)MAX\_SIZE),)

LDPPFLAGS += -DIMAGE\_MAX\_SIZE=$(CONFIG\_$(SPL\_TPL\_)MAX\_SIZE)

endif

那么CONFIG\_SPL\_MAX\_SIZE又是什么值呢？

这个值定义在am65x\_evm.h下面：

#define CONFIG\_SPL\_MAX\_SIZE CONFIG\_SYS\_K3\_MAX\_DOWNLODABLE\_IMAGE\_SIZE

CONFIG\_SYS\_K3\_MAX\_DOWNLOADABLE\_IMAGE\_SIZE即0x58000

所以：

MEMORY { .sram : ORIGIN = 0x80080000,

LENGTH = 0x58000 }

MEMORY { .sdram : ORIGIN = CONFIG\_SPL\_BSS\_START\_ADDR,

LENGTH = CONFIG\_SPL\_BSS\_MAX\_SIZE }

同理可以得出后面这块内存的范围：

MEMORY { .sram : ORIGIN = 0x80080000,

LENGTH = 0x58000 }

MEMORY { .sdram : ORIGIN = 0x80a00000,

LENGTH = 0x80000 }

那这个0x80080000又在哪里呢？



从硬件手册上可以看到，是处于DDR低2G的空间。当需要32bit的地址时候，MSMC会把内存底部的2G映射到0x80000000-0xffffffff地址空间，这样对与DDRHI，低2G不可访问。内存分为2 banks，低2G（32bit），高2G（64bit）。如果不需要32bit的时候，MSMC把所有的内存映射到0x800000000-0xfffffffff这32G空间中。

实际上0x80080000就是在内存的32K地址上。

lds文件后面的内容，其中比较重要的就是：

.text : {

. = ALIGN(8);

\*(.\_\_image\_copy\_start)

CPUDIR/start.o (.text\*)

\*(.text\*)

} >.sram

这节描述.text内容应该放到sram这个内存区域。指定\_image\_copy\_start变量，入口函数是CPUDIR/start.o。其他节分析一样。

有了这些基础，我们开始看u-boot-spl的执行过程：

首先K3是用armv8系列的架构，所以CPUDIR就是arch/arm/cpu/armv8。其中start.o就是start.s编译生成的目标文件。然后入口地址是\_start

start.S分析：

.globl \_start

\_start:

#if defined(CONFIG\_LINUX\_KERNEL\_IMAGE\_HEADER)

#include <asm/boot0-linux-kernel-header.h>

#elif defined(CONFIG\_ENABLE\_ARM\_SOC\_BOOT0\_HOOK)

#include <asm/arch/boot0.h>

#else

b reset

#endif

首先K3并没有定义CONFIG\_LINUX\_KERNEL\_IMAGE\_HEADER以及CONFIG\_ENABLE\_ARM\_SOC\_BOOT0\_HOOK。所以直接执行b reset。

b save\_boot\_params，这是一个弱引用函数。K3并没有特别定义save\_boot\_params，执行start.s里面定义的save\_boot\_params，直接跳转到save\_boot\_param\_ret。由于K3也没有定义CONFIG\_POSITION\_INDEPENDENT，所以运行下面代码:

switch\_el x1, 3f, 2f, 1f

3: set\_vbar vbar\_el3, x0

mrs x0, scr\_el3

orr x0, x0, #0xf /\* SCR\_EL3.NS|IRQ|FIQ|EA \*/

msr scr\_el3, x0

msr cptr\_el3, xzr /\* Enable FP/SIMD \*/

#ifdef COUNTER\_FREQUENCY

ldr x0, =COUNTER\_FREQUENCY

msr cntfrq\_el0, x0 /\* Initialize CNTFRQ \*/

#endif

b 0f

2: set\_vbar vbar\_el2, x0

mov x0, #0x33ff

msr cptr\_el2, x0 /\* Enable FP/SIMD \*/

b 0f

1: set\_vbar vbar\_el1, x0

mov x0, #3 << 20

msr cpacr\_el1, x0 /\* Enable FP/SIMD \*/

0:

isb

这段代码是为了开启ARM v8的EL3/EL2/EL1的运行级别。

其中EL0: applications, EL1: OS kernel, EL2: Hypervisor, EL3: Secure monitor

bl apply\_core\_errata

bl lowlevel\_init

apply\_core\_errata，给CPU打入补丁。

lowlevel\_init同样也是一个弱引用。

弱引用：

如果有下列两个文件：

main.c

test.h

int hello\_world (void);

#include

#include "test.h"

int \_\_attribute\_\_((weak)) hello\_world (void)

{

printf ("weak hello world\n");

return 0;

}

int main (int ac, char \*\*av)

{

hello\_world();

return 0;

}

test.c

#include <stdio.h>

#include "test.h"

int hello\_world (void)

{

printf ("strong hello world\n");

return 0;

}

当我们用gcc test.c main.c得到a.out文件后，执行输出应该是“strong hello world”。

当我们用gcc main.c得到a.out文件后，执行输出“weak hello world”

这就是弱引用的一个范例。当系统中有其他同名函数存在，链接器就是使用没有weak定义的函数。如果没有，那就用weak定义的函数。

K3的lowlevel\_init是使用start.s下面的弱引用：  
从Make file里面看出mach-k3里面的需要定义CONFIG\_CPU\_V7R，而armv8里面是给sunxi使用的。

WEAK(lowlevel\_init)

mov x29, lr /\* Save LR \*/

2:

mov lr, x29 /\* Restore LR \*/

ret

ENDPROC(lowlevel\_init)

\_main:

这部分最重要的就是设置堆栈，C语言执行需要堆栈。

ENTRY(\_main)

#if defined(CONFIG\_TPL\_BUILD) && defined(CONFIG\_TPL\_NEEDS\_SEPARATE\_STACK)

ldr x0, =(CONFIG\_TPL\_STACK)

#elif defined(CONFIG\_SPL\_BUILD) && defined(CONFIG\_SPL\_STACK)

ldr x0, =(CONFIG\_SPL\_STACK)

#elif defined(CONFIG\_INIT\_SP\_RELATIVE)

#if CONFIG\_POSITION\_INDEPENDENT

adrp x0, \_\_bss\_start /\* x0 <- Runtime &\_\_bss\_start \*/

add x0, x0, #:lo12:\_\_bss\_start

#else

adr x0, \_\_bss\_start

#endif

add x0, x0, #CONFIG\_SYS\_INIT\_SP\_BSS\_OFFSET

#else

ldr x0, =(CONFIG\_SYS\_INIT\_SP\_ADDR)

#endif

bic sp, x0, #0xf /\* 16-byte alignment for ABI compliance \*/

mov x0, sp

K3会执行ldr x0, =(CONFIG\_SYS\_INIT\_SP\_ADDR)

#define CONFIG\_SYS\_INIT\_SP\_ADDR (CONFIG\_SPL\_TEXT\_BASE + \

CONFIG\_SYS\_K3\_NON\_SECURE\_MSRAM\_SIZE)

从前面知道CONFIG\_SPL\_TEXT\_BASE=0x80080000，而CONFIG\_SYS\_K3\_NON\_SECURE\_MSRAM\_SIZE定义在.config中，其值为：0x80000。故堆栈的起始位置为0x80100000。sp=0x80100000。

bl board\_init\_f\_alloc\_reserve

ulong board\_init\_f\_alloc\_reserve(ulong top)

{

/\* Reserve early malloc arena \*/

#if CONFIG\_VAL(SYS\_MALLOC\_F\_LEN)

top -= CONFIG\_VAL(SYS\_MALLOC\_F\_LEN);

#endif

/\* LAST : reserve GD (rounded up to a multiple of 16 bytes) \*/

top = rounddown(top-sizeof(struct global\_data), 16);

return top;

}

K3定义了CONFIG\_SPL\_SYS\_MALLOC\_F\_LEN=0x8000。函数参数是x0（即sp）的值：0x80100000。函数退出后，sp指针指向：0x800F8000 – sizeof(gd)。

接下来spl需要给gd分配一块空间，bl board\_init\_f\_reserve。其中的参数还是x0。

mov x18, x0

在整个spl阶段，gd的地址始终保存在x18寄存器中。

#define DECLARE\_GLOBAL\_DATA\_PTR register volatile gd\_t \*gd asm ("x18")

故执行完上面的mov以后，x18就已经是gd的开始地址了。

void board\_init\_f\_init\_reserve(ulong base)

{

struct global\_data \*gd\_ptr;

gd\_ptr = (struct global\_data \*)base;

memset(gd\_ptr, '\0', sizeof(\*gd));

#if !defined(CONFIG\_ARM)

arch\_setup\_gd(gd\_ptr);

#endif

if (CONFIG\_IS\_ENABLED(SYS\_REPORT\_STACK\_F\_USAGE))

board\_init\_f\_init\_stack\_protection\_addr(base);

base += roundup(sizeof(struct global\_data), 16);

#if CONFIG\_VAL(SYS\_MALLOC\_F\_LEN)

/\* go down one 'early malloc arena' \*/

gd->malloc\_base = base;

#endif

if (CONFIG\_IS\_ENABLED(SYS\_REPORT\_STACK\_F\_USAGE))

board\_init\_f\_init\_stack\_protection();

}

前面说过base就是x0的值，也是sp当前的指向0x800F8000。

gd为0x800F8000，初始化。

函数执行完，示意图如下：

+----------+ 0x80100000

+ +

+ +

+ +

+ +

+ +

+ +

+ +

+ +

+----------+0x800F8000 (malloc\_base)

+ + \

+ + the content of gd

+ + /

+----------+0x800F8040 (gd, stack top)

+ +

+ +

+----------+0x80080000 (u\_boot spl)

执行boar\_init\_f，这个board\_init\_f是在SPL阶段所有，我们知道spl阶段能用的空间很小，所以board\_inif\_f不会用board\_f.c里面的函数。而是spl.c下面的弱引用函数。在K3上有自己的定义：arch/arm/mach-k3/am6\_init.c里面的函数：

u32 bootindex \_\_attribute\_\_((section(".data")));

static void store\_boot\_index\_from\_rom(void)

{

bootindex = \*(u32 \*)(CONFIG\_SYS\_K3\_BOOT\_PARAM\_TABLE\_INDEX);

}

为bootindex分配地址，bootindex在.data节。内容在地址0x41c7fbfc。这个地址在



MCU空间。C语言里面这个变量应该在.bss段，但是现在.bss段还没有分配空间，所以把这个变量放到.data段。

ctrl\_mmr\_unlock：解锁所有的WKUP\_CTRL\_MMR0,MCU\_CTRL\_MMR0及CTRL\_MMR0

setup\_am654\_navss\_northbridge：配置NAVSS北桥功能

spl\_early\_init：调用spl\_common\_init，设置GD\_FLG\_SPL\_EARLY\_INIT。

* spl\_common\_init

static int spl\_common\_init(bool setup\_malloc)

{

int ret;

#if CONFIG\_VAL(SYS\_MALLOC\_F\_LEN)

if (setup\_malloc) {

gd->malloc\_limit = CONFIG\_VAL(SYS\_MALLOC\_F\_LEN);

gd->malloc\_ptr = 0;

}

#endif

我们从.config文件里面得知：CONFIG\_SPL\_SYS\_MALLOC\_F\_LEN=0x8000，是32K的空间。前面分配的高地址是0x80100000,malloc\_base=0x800F8000，刚好是32K空间。

ret = bootstage\_init(u\_boot\_first\_phase());

if (ret) {

debug("%s: Failed to set up bootstage: ret=%d\n", \_\_func\_\_,

ret);

return ret;

}

bootstage\_mark\_name(spl\_phase() == PHASE\_TPL ? BOOTSTAGE\_ID\_START\_TPL :

BOOTSTAGE\_ID\_START\_SPL, SPL\_TPL\_NAME);

#if CONFIG\_IS\_ENABLED(LOG)

ret = log\_init();

if (ret) {

debug("%s: Failed to set up logging\n", \_\_func\_\_);

return ret;

}

#endif

K3定义了SYS\_MALLOC\_SIMPLE，所以这里用的malloc是定义在include/malloc.h

#define malloc malloc\_simple

void \*malloc\_simple(size\_t bytes)

{

void \*ptr;

ptr = alloc\_simple(bytes, 1);

if (!ptr)

return ptr;

log\_debug("%lx\n", (ulong)ptr);

return ptr;

}

即在前面32K处分配空间。

K3没有定义CONFIG\_BOOTSTAGE，也没有定义CONFIG\_SPL\_LOG，此部分不执行。

if (CONFIG\_IS\_ENABLED(OF\_CONTROL) && !CONFIG\_IS\_ENABLED(OF\_PLATDATA)) {

ret = fdtdec\_setup();

if (ret) {

debug("fdtdec\_setup() returned error %d\n", ret);

return ret;

}

}

K3定义了CONFIG\_SPL\_OF\_CONTROL，但没有定义OF\_PLATDATA。所以执行fdtdec\_setup，为了加载device tree的信息。

加载的地址在gd->fdt\_blob，下面看函数fdtdec\_setup：

执行board\_fdt\_blob\_setup函数，这是一个弱函数。K3没有定义这个函数，使用lib/fdtdec.c定义的弱函数。

int fdtdec\_setup(void)

{

int ret;

#if CONFIG\_IS\_ENABLED(OF\_CONTROL)

# if CONFIG\_IS\_ENABLED(MULTI\_DTB\_FIT)

void \*fdt\_blob;

# endif

# if defined(CONFIG\_OF\_BOARD) || defined(CONFIG\_OF\_SEPARATE)

gd->fdt\_blob = board\_fdt\_blob\_setup();

# endif

# if CONFIG\_IS\_ENABLED(MULTI\_DTB\_FIT)

if (uncompress\_blob(gd->fdt\_blob, 0x1000000, &fdt\_blob) == 0)

gd->fdt\_blob = fdt\_blob;

fdt\_blob = locate\_dtb\_in\_fit(gd->fdt\_blob);

if (fdt\_blob) {

gd->multi\_dtb\_fit = gd->fdt\_blob;

gd->fdt\_blob = fdt\_blob;

}

# endif

#endif

ret = fdtdec\_prepare\_fdt();

if (!ret)

ret = fdtdec\_board\_setup(gd->fdt\_blob);

return ret;

}

我们来看下这个函数实现：

前面我们看到在K3镜像上dtb放在spl镜像的后面，所以只需要把\_image\_binary\_end赋值给fdt\_blob即可，后面的内容即为dtb的内容。

\_\_weak void \*board\_fdt\_blob\_setup(void)

{

void \*fdt\_blob = NULL;

#ifdef CONFIG\_SPL\_BUILD

/\* FDT is at end of BSS unless it is in a different memory region \*/

if (IS\_ENABLED(CONFIG\_SPL\_SEPARATE\_BSS))

fdt\_blob = (ulong \*)&\_image\_binary\_end;

else

fdt\_blob = (ulong \*)&\_\_bss\_end;

#else

/\* FDT is at end of image \*/

fdt\_blob = (ulong \*)&\_end;

#endif

return fdt\_blob;

}

K3的dtb不经过压缩，跳过uncompress\_blob。

locate\_dtb\_in\_fit，查找FIT的头部，计算真正的fdt地址。

fdtdec\_prepare\_fdt，检查dtb头部是否正确。

这里定义了CONFIG\_SPL\_DM，执行dm\_init\_and\_scan，OF\_PLATDATA并没有定义，所以参数为TRUE

if (CONFIG\_IS\_ENABLED(DM)) {

bootstage\_start(BOOTSTAGE\_ID\_ACCUM\_DM\_SPL,

spl\_phase() == PHASE\_TPL ? "dm tpl" : "dm\_spl");

/\* With CONFIG\_SPL\_OF\_PLATDATA, bring in all devices \*/

ret = dm\_init\_and\_scan(!CONFIG\_IS\_ENABLED(OF\_PLATDATA));

bootstage\_accum(BOOTSTAGE\_ID\_ACCUM\_DM\_SPL);

if (ret) {

debug("dm\_init\_and\_scan() returned error %d\n", ret);

return ret;

}

}

这函数主要是初始化硬件驱动。U-boot的驱动是以

int dm\_init\_and\_scan(bool pre\_reloc\_only)

{

int ret;

if (CONFIG\_IS\_ENABLED(OF\_PLATDATA))

dm\_populate\_phandle\_data();

ret = dm\_init(CONFIG\_IS\_ENABLED(OF\_LIVE));

if (ret) {

debug("dm\_init() failed: %d\n", ret);

return ret;

}

ret = dm\_scan\_platdata(pre\_reloc\_only);

if (ret) {

debug("dm\_scan\_platdata() failed: %d\n", ret);

return ret;

}

if (CONFIG\_IS\_ENABLED(OF\_CONTROL) && !CONFIG\_IS\_ENABLED(OF\_PLATDATA)) {

ret = dm\_extended\_scan\_fdt(gd->fdt\_blob, pre\_reloc\_only);

if (ret) {

debug("dm\_extended\_scan\_dt() failed: %d\n", ret);

return ret;

}

}

ret = dm\_scan\_other(pre\_reloc\_only);

if (ret)

return ret;

return 0;

}

Ll\_entry\_declare声明驱动的定义，放入.u\_boot\_list\_2这个专用段。

#define ll\_entry\_declare(\_type, \_name, \_list) \

\_type \_u\_boot\_list\_2\_##\_list##\_2\_##\_name \_\_aligned(4) \

\_\_attribute\_\_((unused, \

section(".u\_boot\_list\_2\_"#\_list"\_2\_"#\_name)))

驱动初始化后，配置串口的信息preloader\_console\_init：

void preloader\_console\_init(void)

{

gd->baudrate = CONFIG\_BAUDRATE;

serial\_init(); /\* serial communications setup \*/

gd->have\_console = 1;

}

int serial\_init(void)

{

gd->flags |= GD\_FLG\_SERIAL\_READY;

return get\_current()->start();

}

设置串口波特率gd->baudrate = 115200，打开串口时钟，激活当前串口。

K3没有定义CONFIG\_SYS\_ICACHE\_OFF和CONFIG\_SYS\_DCAHE\_OFF。 CONFIG\_SYS\_SDRAM\_BASE=0x80000000，硬件手册上已经描述了。这里值得注意的是dram\_init\_banksize函数。

int dram\_init\_banksize(void)

{

/\* Bank 0 declares the memory available in the DDR low region \*/

gd->bd->bi\_dram[0].start = CONFIG\_SYS\_SDRAM\_BASE;

gd->bd->bi\_dram[0].size = 0x80000000;

gd->ram\_size = 0x80000000;

#ifdef CONFIG\_PHYS\_64BIT

/\* Bank 1 declares the memory available in the DDR high region \*/

gd->bd->bi\_dram[1].start = CONFIG\_SYS\_SDRAM\_BASE1;

gd->bd->bi\_dram[1].size = 0x80000000;

gd->ram\_size = 0x100000000;

#endif

return 0;

}

void spl\_enable\_dcache(void)

{

#if !(defined(CONFIG\_SYS\_ICACHE\_OFF) && defined(CONFIG\_SYS\_DCACHE\_OFF))

phys\_addr\_t ram\_top = CONFIG\_SYS\_SDRAM\_BASE;

dram\_init\_banksize();

/\* reserve TLB table \*/

gd->arch.tlb\_size = PGTABLE\_SIZE;

ram\_top += get\_effective\_memsize();

/\* keep ram\_top in the 32-bit address space \*/

if (ram\_top >= 0x100000000)

ram\_top = (phys\_addr\_t) 0x100000000;

gd->arch.tlb\_addr = ram\_top - gd->arch.tlb\_size;

debug("TLB table from %08lx to %08lx\n", gd->arch.tlb\_addr,

gd->arch.tlb\_addr + gd->arch.tlb\_size);

dcache\_enable();

#endif

}

内存设置了两个bank， 第一个bank起始位置在0x80000000， 大小为2G。第二个bank起始位置在0x880000000，大小以为2G。总共内存大小4G。正好跟硬件手册上描述一下，bank[0]供32bit应用使用。Bank[1]只能是64bit应用使用。

为TLB table保留地址，arm64使用get\_page\_table\_size函数得到。

我们K3的memory size是0x100000000，所以tlb\_addr就在memory顶端处， 示意如下：

+----------+ 0x980000000

+ +

+ +

+ +

+ +

+ +

+ +

+ +

+ +

+----------+ 0x97ffff000 (gd->arch.tlb\_addr)

打开dcache支持。

执行完board\_init\_f后，继续回到crt0\_64.s

接下来执行spl\_relocate\_stack\_gd

ulong spl\_relocate\_stack\_gd(void)

{

#ifdef CONFIG\_SPL\_STACK\_R

gd\_t \*new\_gd;

ulong ptr = CONFIG\_SPL\_STACK\_R\_ADDR;

#if defined(CONFIG\_SPL\_SYS\_MALLOC\_SIMPLE) && CONFIG\_VAL(SYS\_MALLOC\_F\_LEN)

if (CONFIG\_SPL\_STACK\_R\_MALLOC\_SIMPLE\_LEN) {

ptr -= CONFIG\_SPL\_STACK\_R\_MALLOC\_SIMPLE\_LEN;

gd->malloc\_base = ptr;

gd->malloc\_limit = CONFIG\_SPL\_STACK\_R\_MALLOC\_SIMPLE\_LEN;

gd->malloc\_ptr = 0;

}

#endif

/\* Get stack position: use 8-byte alignment for ABI compliance \*/

ptr = CONFIG\_SPL\_STACK\_R\_ADDR - roundup(sizeof(gd\_t),16);

new\_gd = (gd\_t \*)ptr;

memcpy(new\_gd, (void \*)gd, sizeof(gd\_t));

#if CONFIG\_IS\_ENABLED(DM)

dm\_fixup\_for\_gd\_move(new\_gd);

#endif

return ptr;

#endif

}

这个函数为gd分配一个新的地址CONFIG\_SPL\_STACK\_R\_ADDR（0x82000000）。

为malloc分配0x100000的大小。接下来就是new gd的空间。

+----------+ 0x82000000

+ +

+ +

+----------+ 0x81ffffc0 (new\_gd)

+ +

+ +

+----------+ 0x81f00000

把新的new\_gd地址保存在x18中，然后清空BSS段。BSS段在0x80a00000的位置。

.bss\_start (NOLOAD) : {

. = ALIGN(8);

KEEP(\*(.\_\_bss\_start));

} >.sdram

.bss (NOLOAD) : {

\*(.bss\*)

. = ALIGN(8);

} >.sdram

.bss\_end (NOLOAD) : {

KEEP(\*(.\_\_bss\_end));

} >.sdram

前文说过sdram的地址就是0x80a00000，大小为512K

+----------+ 0x82000000

+ +

+ +

+----------+ 0x81ffffc0 (new\_gd)

+ +

+ +

+----------+ 0x81f00000

+ +

+ +

+ +

+----------+ 0x80a00000 (BSS)

调用board\_init\_r(common/spl.c)。

在board\_init\_r里面会通过spl\_boot\_device取得启动设备，K3 u-boot在MMC上，就返回BOOT\_DEVICE\_MMC1。

spl在board\_init\_f已经初始化了dm。这里就能够使用驱动了。在boot\_from\_devices里面：会使用已经声明在.uboot\_list\_2\_段的驱动。如果从MMC设备找镜像，那就使用spl\_mmc.c下面的spl\_mmc\_load\_image。

MMC镜像存储方式：

+----------+ 0x0 - tiboot3.bin

+ +

+ +

+----------+ 0x400 – tispl.bin

+ +

+ +

+----------+ 0x1400 – u-boot.img

+ +

+ +

+----------+ 0x3600 – ?

Board\_init\_r把u-boot镜像复制到0x80800000的位置，然后跳转到u-boot镜像。

u-boot spl结束。

**U-boot阶段：**

+----------+ 0x82000000

+ +

+ +

+----------+ 0x81ffffc0 (new\_gd, assume gd size is 0x40)

+ +

+ +

+ +

+ +

+----------+ 0x80a00000 (spl BSS)

+ + \

+ + 2M memory spaces

+ + /

+----------+ 0x80800000 (U-boot)

这是U-boot在K3起来时的内存镜像。

spl已经结束，此时u-boot从0x80800000处开始运行。

又来到熟悉的start.s文件。这个阶段已经是完全体的u-boot了。

u-boot启动后加载.vectors中断向量表，arch/arm/cpu/armv8/exceptions.s。然后根据arm的运行级别（EL0,1,2,3）设置到相应的中断处理例程set\_vbar vbar\_el2, x0。

bl apply\_core\_errata 为core打入errata补丁。

bl lowlevel\_init跟spl一样，什么都不干。

跳转到\_main，流程跟spl一样，同样这时时候堆栈其实也一样。

+----------+ 0x80100000

+ +

+ +

+ +

+ +

+ +

+ +

+ +

+ +

+----------+0x800F8000 (malloc\_base)

+ + \

+ + the content of gd

+ + /

+----------+0x800F8040 (gd, stack top)

接下来要处理的就是u-boot的重头戏board\_init\_f以及board\_init\_r两个函数，为最后把控制权交给Linux kernel做准备。这时候的board\_init\_f是位于board\_f.c下面。

跳转过来的时候boot\_flags设置为0。故gd->flags=0。

void board\_init\_f(ulong boot\_flags)

{

gd->flags = boot\_flags;

gd->have\_console = 0;

if (initcall\_run\_list(init\_sequence\_f))

hang();

}

接下来就是执行init\_sequence\_f定义的函数。

static int setup\_mon\_len(void)

{

gd->mon\_len = (ulong)&\_\_bss\_end - (ulong)\_start;

return 0;

}

fdtdec\_setup跟spl处理一样，最终gd->fdt\_blob记录dt的地址。唯一的不同是这里的dt是从u-boot镜像\_end处得到的。

initf\_malloc设置malloc的长度。

arch\_cpu\_init 弱函数类型，直接return

mach\_cpu\_init 跟arch\_cpu\_init一样

initf\_dm 初始化驱动，跟spl阶段一样。

arch\_cpu\_init\_dm 弱函数，直接return。

timer\_init 弱函数，直接return。

env\_init 查找ENV的驱动，由U\_BOOT\_ENV\_LOCATION声明，同样定义在.uboot\_list\_2\_段。

env\_driver\_lookup查找驱动，K3在MMC上由CONFIG\_ENV\_IS\_IN\_MMC=y得到，

int env\_init(void)

{

struct env\_driver \*drv;

int ret = -ENOENT;

int prio;

for (prio = 0; (drv = env\_driver\_lookup(ENVOP\_INIT, prio)); prio++) {

if (!drv->init || !(ret = drv->init()))

env\_set\_inited(drv->location);

if (ret == -ENOENT)

env\_set\_inited(drv->location);

debug("%s: Environment %s init done (ret=%d)\n", \_\_func\_\_,

drv->name, ret);

}

if (!prio)

return -ENODEV;

if (ret == -ENOENT) {

gd->env\_addr = (ulong)&default\_environment[0];

gd->env\_valid = ENV\_VALID;

return 0;

}

return ret;

}

最后设置gd->env\_addr的地址为default\_environment

U\_BOOT\_ENV\_LOCATION(mmc) = {

.location = ENVL\_MMC,

ENV\_NAME("MMC")

.load = env\_mmc\_load,

#ifndef CONFIG\_SPL\_BUILD

.save = env\_save\_ptr(env\_mmc\_save),

#if defined(CONFIG\_CMD\_ERASEENV)

.erase = env\_mmc\_erase,

#endif

#endif

};

init\_baud\_rate 设置串口波特率，K3为115200，gd->baudrate

serial\_init 初始化串口。

console\_init\_f console初始化，gd->have\_console为1，因为K3没有定义PRE\_CONSOLE\_BUFFER，所以这里不输出buffer。

display\_text\_info 输出text\_base, bss\_start, bss\_end信息。text\_base 0x80800000。

print\_resetinfo

print\_cpuinfo

show\_board\_info, 打印reset，cpu，board的信息，不做具体深入分析。

K3 uboot没有定义看门狗？？？

K3有DM\_I2C，不执行init\_func\_i2c

int dram\_init(void)

{

#ifdef CONFIG\_PHYS\_64BIT

gd->ram\_size = 0x100000000;

#else

gd->ram\_size = 0x80000000;

#endif

return 0;

}

很明显ram\_size是4G，K3硬件手册也是定义为4G

setup\_dest\_addr

static int setup\_dest\_addr(void)

{

debug("Monitor len: %08lX\n", gd->mon\_len);

debug("Ram size: %08lX\n", (ulong)gd->ram\_size);

#ifdef CONFIG\_SYS\_SDRAM\_BASE

gd->ram\_base = CONFIG\_SYS\_SDRAM\_BASE;

#endif

gd->ram\_top = gd->ram\_base + get\_effective\_memsize();

gd->ram\_top = board\_get\_usable\_ram\_top(gd->mon\_len);

gd->relocaddr = gd->ram\_top;

debug("Ram top: %08lX\n", (ulong)gd->ram\_top);

return 0;

}

gd->ram\_base为0x80000000，经过计算后gd->ram\_top=0x100000000。刚好是低2G DDR内存的最高位。同时gd->relocaddr也在这个位置，这个地址是后面重新分配空间所需要用到。

reserve\_round\_4k 保留最高4K的地址，这里操作的是relocaddr，ram\_top值并没有改变。

+----------+ 0xffffffff

+ +

+----------+ 0Xfffff000 (reserve 4K)

arm\_reserve\_mmu 设置MMU TLB表空间，

等于：512\*8\*6\*2 + 512\*8\*4 = 65536.

\_\_weak int arm\_reserve\_mmu(void)

{

gd->arch.tlb\_size = PGTABLE\_SIZE;

gd->relocaddr -= gd->arch.tlb\_size;

gd->relocaddr &= ~(0x10000 - 1);

gd->arch.tlb\_addr = gd->relocaddr;

return 0;

}

gd->relocaddr=0xfffe0000。

+----------+ 0xffffffff

+ +

+----------+ 0Xfffff000 (reserve 4K)

+ +

+----------+ 0Xfffe0000 (gd->arch.tlb\_addr)

reserve\_uboot 给ubootreserve空间。

+----------+ 0xffffffff

+ +

+----------+ 0xfffff000 (reserve 4K)

+ +

+----------+ 0xfffe0000 (gd->arch.tlb\_addr)

+ + \

+ + BSS size

+ + /

+----------+ 0x????????

+ + 4K reserve

+----------+ gd->start\_addr\_sp

继续下面reserve\_malloc， reserve\_board：

+----------+ 0xffffffff

+ +

+----------+ 0xfffff000 (reserve 4K)

+ +

+----------+ 0xfffe0000 (gd->arch.tlb\_addr)

+ + \

+ + BSS size

+ + /

+----------+ 0x????????

+ + 4K reserve

+----------+

+ + malloc len

+----------+

+ + bdinfo

+----------+ gd->start\_addr\_sp

reserve\_global\_data: 这里重新分配gd的空间。

static int reserve\_global\_data(void)

{

gd->start\_addr\_sp = reserve\_stack\_aligned(sizeof(gd\_t));

gd->new\_gd = (gd\_t \*)map\_sysmem(gd->start\_addr\_sp, sizeof(gd\_t));

debug("Reserving %zu Bytes for Global Data at: %08lx\n",

sizeof(gd\_t), gd->start\_addr\_sp);

return 0;

}

+----------+ 0xffffffff

+ +

+----------+ 0xfffff000 (reserve 4K)

+ +

+----------+ 0xfffe0000 (gd->arch.tlb\_addr)

+ + \

+ + BSS size

+ + /

+----------+ 0x????????

+ + 4K reserve

+----------+

+ + malloc len

+----------+

+ + bdinfo

+----------+

+ + gd->new\_gd

+----------+ gd->start\_addr\_sp

reserve\_stacks后：

+----------+ 0xffffffff

+ +

+----------+ 0xfffff000 (reserve 4K)

+ +

+----------+ 0xfffe0000 (gd->arch.tlb\_addr)

+ + \

+ + BSS size

+ + /

+----------+ 0x????????

+ + 4K reserve

+----------+

+ + malloc len

+----------+

+ + bdinfo

+----------+

+ + gd->new\_gd

+----------+

+ + reserve 16 bytes

+----------+ gd->start\_addr\_sp (gd->irq\_sp)

这里设置了中断的堆栈。

dram\_init\_banksize

K3有两个bank。从0x80000000地址开始，大小2G，和0x880000000地址开始，大小2G

int dram\_init\_banksize(void)

{

gd->bd->bi\_dram[0].start = CONFIG\_SYS\_SDRAM\_BASE;

gd->bd->bi\_dram[0].size = 0x80000000;

gd->ram\_size = 0x80000000;

#ifdef CONFIG\_PHYS\_64BIT

/\* Bank 1 declares the memory available in the DDR high region \*/

gd->bd->bi\_dram[1].start = CONFIG\_SYS\_SDRAM\_BASE1;

gd->bd->bi\_dram[1].size = 0x80000000;

gd->ram\_size = 0x100000000;

#endif

return 0;

}

setup\_reloc： 这里计算u-boot到relocaddr的偏移，计算结果保存到gd->reloc\_off。拷贝gd的内容到gd->new\_gd，图上的位置。

至此board\_init\_f执行完毕，回到crt0\_64.s下面。

#if !defined(CONFIG\_SPL\_BUILD)

ldr x0, [x18, #GD\_START\_ADDR\_SP] /\* x0 <- gd->start\_addr\_sp \*/

bic sp, x0, #0xf /\* 16-byte alignment for ABI compliance \*/

ldr x18, [x18, #GD\_NEW\_GD] /\* x18 <- gd->new\_gd \*/

adr lr, relocation\_return

ldr x9, [x18, #GD\_RELOC\_OFF] /\* x9 <- gd->reloc\_off \*/

add lr, lr, x9 /\* new return address after relocation \*/

ldr x0, [x18, #GD\_RELOCADDR] /\* x0 <- gd->relocaddr \*/

b relocate\_code

relocation\_return:

bl c\_runtime\_cpu\_setup /\* still call old routine \*/

#endif

这段汇编主要是把gd->start\_addr\_sp地址赋值给x0，前面说过这个地址也是中断堆栈的地址。把这个地址按16字节对齐后送入sp。设置new gd的地址给x18。

记录退回地址。前面计算的偏移量保存到x9。relocaddr送给x0。调用relocate\_code。

relocate\_code主要把u-boot整个镜像拷贝到relocaddr处：

+----------+ 0xffffffff

+ +

+----------+ 0xfffff000 (reserve 4K)

+ +

+----------+ 0xfffe0000 (gd->arch.tlb\_addr)

+ + \

+ + u-boot

+ + /

+----------+ 0x????????

+ + 4K reserve

+----------+

+ + malloc len

+----------+

+ + bdinfo

+----------+

+ + gd->new\_gd

+----------+

+ + reserve 16 bytes

+----------+ gd->start\_addr\_sp (gd->irq\_sp)

最后清零bss段，准备board\_init\_r的两个参数gd和relocaddr。调用u-boot生命里面最后一个函数：

board\_init\_r，这个函数永远不会返回。成功后就开始运行kernel，不成功整个K3板子就此陷入死循环。

函数主要按顺序执行在init\_sequence\_r里面所定义的函数。

static int initr\_reloc(void)

{

/\* tell others: relocation done \*/

gd->flags |= GD\_FLG\_RELOC | GD\_FLG\_FULL\_MALLOC\_INIT;

return 0;

}

initr\_caches打开ICACHE和DCACHE支持。

static int initr\_caches(void)

{

/\* Enable caches \*/

enable\_caches();

return 0;

}

initr\_reloc\_global\_data这个函数主要是对一些全局变量的地址进行重新计算，因为前面u-boot已经搬过一次家了，所以这些地址就要加上reloc\_off。

initr\_malloc 分配malloc的空间。

initr\_dm 驱动初始化。

efi\_memory\_init 给EFI内存初始化，不做具体深入，这是EFI boot需要用到，最终就是带起Linux kernel

接下来比较有趣的是我们要支持PCI/MMC？那么就相应地调用initr\_pci, initr\_mmc等特有的驱动函数。

initr\_env 这里比较重要的一点就是会增加fdtcontroladdr字段到dt，然后从环境变量里面读出loadaddr的值放到image\_load\_addr。这个地址是为了把操作系统镜像加载到内存的地方，注意只是加载，并不是说从这个地址开始执行。K3里面是0x82000000。

余下一些函数并不是太重要，就不一一展开。执行完后就会跳到run\_main\_loop函数。

这是一个大循环函数，里面接受用户输入命令或执行已经定义命令。

K3定义的命令是

run findfdt; run envboot; run init\_${boot}; run boot\_rprocs; run get\_kern\_${boot}; run get\_fdt\_${boot}; run get\_overlay\_${boot}; run run\_kern

void main\_loop(void)

{

const char \*s;

bootstage\_mark\_name(BOOTSTAGE\_ID\_MAIN\_LOOP, "main\_loop");

if (IS\_ENABLED(CONFIG\_VERSION\_VARIABLE))

env\_set("ver", version\_string); /\* set version variable \*/

cli\_init();

if (IS\_ENABLED(CONFIG\_USE\_PREBOOT))

run\_preboot\_environment\_command();

if (IS\_ENABLED(CONFIG\_UPDATE\_TFTP))

update\_tftp(0UL, NULL, NULL);

s = bootdelay\_process();

if (cli\_process\_fdt(&s))

cli\_secure\_boot\_cmd(s);

autoboot\_command(s);

cli\_loop();

panic("No CLI available");

}

可以看到首先执行的是autoboot\_command。如果没有才会进入命令行，接受用户的输入。

首先拿到bootdelay，也就是u-boot启动kernel的时候有个倒计时，就是这个值。这个值如果在dt中定义了，那就优先使用dt里面的。

const char \*bootdelay\_process(void)

{

char \*s;

int bootdelay;

bootcount\_inc();

s = env\_get("bootdelay");

bootdelay = s ? (int)simple\_strtol(s, NULL, 10) : CONFIG\_BOOTDELAY;

if (IS\_ENABLED(CONFIG\_OF\_CONTROL))

bootdelay = fdtdec\_get\_config\_int(gd->fdt\_blob, "bootdelay",

bootdelay);

debug("### main\_loop entered: bootdelay=%d\n\n", bootdelay);

if (IS\_ENABLED(CONFIG\_AUTOBOOT\_MENU\_SHOW))

bootdelay = menu\_show(bootdelay);

bootretry\_init\_cmd\_timeout();

#ifdef CONFIG\_POST

if (gd->flags & GD\_FLG\_POSTFAIL) {

s = env\_get("failbootcmd");

} else

#endif /\* CONFIG\_POST \*/

if (bootcount\_error())

s = env\_get("altbootcmd");

else

s = env\_get("bootcmd");

if (IS\_ENABLED(CONFIG\_OF\_CONTROL))

process\_fdt\_options(gd->fdt\_blob);

stored\_bootdelay = bootdelay;

return s;

}

从环境变量中拿出bootcmd。位autoboot\_command函数做准备。

K3的boot有几个阶段，其中get\_kern\_${boot}，即get\_kern\_mmc。加载kernel镜像到0x82000000的地址，接下来加载linux的DT到指定的地址。最后运行run\_kern，即booti ${loadaddr} ${rd\_spec} ${fdtaddr}

我们来看booti，这是由U\_BOOT\_CMD宏定义声明的。

这个函数就是为了启动Image进行最后的处理。

static int booti\_start(struct cmd\_tbl \*cmdtp, int flag, int argc,

char \*const argv[], bootm\_headers\_t \*images)

ret = do\_bootm\_states(cmdtp, flag, argc, argv, BOOTM\_STATE\_START,

images, 1);

int do\_booti(struct cmd\_tbl \*cmdtp, int flag, int argc, char \*const argv[])

{

int ret;

/\* Consume 'booti' \*/

argc--; argv++;

if (booti\_start(cmdtp, flag, argc, argv, &images))

return 1;

/\*

\* We are doing the BOOTM\_STATE\_LOADOS state ourselves, so must

\* disable interrupts ourselves

\*/

bootm\_disable\_interrupts();

images.os.os = IH\_OS\_LINUX;

images.os.arch = IH\_ARCH\_ARM64;

ret = do\_bootm\_states(cmdtp, flag, argc, argv,

BOOTM\_STATE\_RAMDISK |

BOOTM\_STATE\_OS\_PREP | BOOTM\_STATE\_OS\_FAKE\_GO |

BOOTM\_STATE\_OS\_GO,

&images, 1);

return ret;

}

这里会调用do\_bootm\_states函数，我们来看下这个函数，因为给出的参数是BOOTM\_STATE\_START，所以执行bootm\_start：

首先清空images这个image header结构体。然后初始化逻辑内存块（lmb）。lmb主要是根据bootm\_low和bootm\_size来设置。其中K3的bootm\_size是256M。退出后执行booti\_setup

ret = booti\_setup(ld, &relocated\_addr, &image\_size, false);

if (ret != 0)

return 1;

static int bootm\_start(struct cmd\_tbl \*cmdtp, int flag, int argc,

char \*const argv[])

{

memset((void \*)&images, 0, sizeof(images));

images.verify = env\_get\_yesno("verify");

boot\_start\_lmb(&images);

bootstage\_mark\_name(BOOTSTAGE\_ID\_BOOTM\_START, "bootm\_start");

images.state = BOOTM\_STATE\_START;

return 0;

}

booti\_setup 主要检查kernel的魔术数，得到kernel的大小和加载地址。因为K3使用random address，所以我们假定加载地址为：0x1e0000，同时计算出重新映射的地址，这个地址需要2M对齐。

if (relocated\_addr != ld) {

printf("Moving Image from 0x%lx to 0x%lx, end=%lx\n", ld,

relocated\_addr, relocated\_addr + image\_size);

memmove((void \*)relocated\_addr, (void \*)ld, image\_size);

}

如果重新映射地址跟加载地址不一样，那么把镜像拷贝到新的映射地址。

images->ep = relocated\_addr;

images->os.start = relocated\_addr;

images->os.end = relocated\_addr + image\_size;

lmb\_reserve(&images->lmb, images->ep, le32\_to\_cpu(image\_size));

if (bootm\_find\_images(flag, argc, argv, relocated\_addr, image\_size))

return 1;

确定kernel的入口地址，镜像大小。Reserve LMB内存。

bootm\_find\_images，简单分析一下：

int bootm\_find\_images(int flag, int argc, char \*const argv[], ulong start,

ulong size)

{

int ret;

ret = boot\_get\_ramdisk(argc, argv, &images, IH\_INITRD\_ARCH,

&images.rd\_start, &images.rd\_end);

if (ret) {

puts("Ramdisk image is corrupt or invalid\n");

return 1;

}

if (images.rd\_start && (((ulong)images.rd\_start >= start &&

(ulong)images.rd\_start < start + size) ||

((ulong)images.rd\_end > start &&

(ulong)images.rd\_end <= start + size) ||

((ulong)images.rd\_start < start &&

(ulong)images.rd\_end >= start + size))) {

printf("ERROR: RD image overlaps OS image (OS=0x%lx..0x%lx)\n",

start, start + size);

return 1;

}

#if IMAGE\_ENABLE\_OF\_LIBFDT

ret = boot\_get\_fdt(flag, argc, argv, IH\_ARCH\_DEFAULT, &images,

&images.ft\_addr, &images.ft\_len);

if (ret) {

puts("Could not find a valid device tree\n");

return 1;

}

if (images.ft\_addr &&

(((ulong)images.ft\_addr >= start &&

(ulong)images.ft\_addr <= start + size) ||

((ulong)images.ft\_addr + images.ft\_len >= start &&

(ulong)images.ft\_addr + images.ft\_len <= start + size))) {

printf("ERROR: FDT image overlaps OS image (OS=0x%lx..0x%lx)\n",

start, start + size);

return 1;

}

if (CONFIG\_IS\_ENABLED(CMD\_FDT))

set\_working\_fdt\_addr(map\_to\_sysmem(images.ft\_addr));

#endif

#if IMAGE\_ENABLE\_FIT

/\* find all of the loadables \*/

ret = boot\_get\_loadable(argc, argv, &images, IH\_ARCH\_DEFAULT,

NULL, NULL);

if (ret) {

printf("Loadable(s) is corrupt or invalid\n");

return 1;

}

#endif

return 0;

}

首先取得ramdisk的镜像，也就是我们的rootfs。找到后如果FIT有指定加载地址，不然就使用0x82000000。前面内核已经搬过家了，所以现在这个地址等于可以重新使用。

检查ramdisk的空间是否跟内核重叠？如果重叠，就此打断自动启动。

接下来取出dtb文件，加载到指定位置。检查是否跟OS image地址重叠。

设置fdtaddr环境变量。

boot\_get\_loadable，找出所有的load entry。

如果上面都没有发生问题，实际上也不会有问题。那么接下来关闭所有中断bootm\_disable\_interrupts()。因为接下来是要启动内核了，是个相当神圣的时候，不容许任何外部的干扰，哪怕总统来了也不行。

images.os.os = IH\_OS\_LINUX;

images.os.arch = IH\_ARCH\_ARM64;

ret = do\_bootm\_states(cmdtp, flag, argc, argv,

BOOTM\_STATE\_RAMDISK |

BOOTM\_STATE\_OS\_PREP | BOOTM\_STATE\_OS\_FAKE\_GO |

BOOTM\_STATE\_OS\_GO,

&images, 1);

state是一个组合，前面do\_bootm\_states已经处理过BOOTM\_STATE\_START了。现在先进行处理的是：BOOTM\_STATE\_RAMDISK（按顺序来）

移动ramdisk到指定地址，这个地址由initrd\_high 或 bootm\_row + bootm\_size指定。

boot\_fn = bootm\_os\_get\_boot\_func(images->os.os);

#ifdef CONFIG\_SYS\_BOOT\_RAMDISK\_HIGH

if (!ret && (states & BOOTM\_STATE\_RAMDISK)) {

ulong rd\_len = images->rd\_end - images->rd\_start;

ret = boot\_ramdisk\_high(&images->lmb, images->rd\_start,

rd\_len, &images->initrd\_start, &images->initrd\_end);

if (!ret) {

env\_set\_hex("initrd\_start", images->initrd\_start);

env\_set\_hex("initrd\_end", images->initrd\_end);

}

}

#endif

取得启动kernel的函数，该函数由boot\_os函数数组指定，我们需要Linux的启动函数：

static boot\_os\_fn \*boot\_os[] = {

#ifdef CONFIG\_BOOTM\_LINUX

[IH\_OS\_LINUX] = do\_bootm\_linux,

#endif

};

前面已经指定了image.os.os为IH\_OS\_LINUX。这个得到do\_bootm\_linux

然后执行do\_bootm\_linux (BOOTM\_STATE\_OS\_PREP, argc, argv, images)-> boot\_prep\_linux(images);

这个函数相当重要。需要详细讲一下：

进入函数后实际上我们执行的是image.c下面的image\_setup\_linux函数。

image\_setup\_linux

|- boot\_fdt\_add\_mem\_rsv\_regions

|- boot\_relocate\_fdt

|- image\_setup\_libfdt

这三个函数最重要的是最后一个函数，前面两个简单来讲就是为了给dtb文件分配空间，搬移到新的空间。

限于篇幅，就不再深入了。我们重点讲下第三个函数image\_setup\_libfdt

fdt\_root

int image\_setup\_libfdt(bootm\_headers\_t \*images, void \*blob,

int of\_size, struct lmb \*lmb)

{

ulong \*initrd\_start = &images->initrd\_start;

ulong \*initrd\_end = &images->initrd\_end;

int ret = -EPERM;

int fdt\_ret;

if (fdt\_root(blob) < 0) {

printf("ERROR: root node setup failed\n");

goto err;

}

if (fdt\_chosen(blob) < 0) {

printf("ERROR: /chosen node create failed\n");

goto err;

}

if (arch\_fixup\_fdt(blob) < 0) {

printf("ERROR: arch-specific fdt fixup failed\n");

goto err;

}

fdt\_ret = optee\_copy\_fdt\_nodes(gd->fdt\_blob, blob);

if (fdt\_ret) {

printf("ERROR: transfer of optee nodes to new fdt failed: %s\n",

fdt\_strerror(fdt\_ret));

goto err;

}

fdt\_fixup\_ethernet(blob);

if (IMAGE\_OF\_BOARD\_SETUP) {

fdt\_ret = ft\_board\_setup(blob, gd->bd);

if (fdt\_ret) {

printf("ERROR: board-specific fdt fixup failed: %s\n",

fdt\_strerror(fdt\_ret));

goto err;

}

}

if (lmb)

lmb\_free(lmb, (phys\_addr\_t)(u32)(uintptr\_t)blob,

(phys\_size\_t)fdt\_totalsize(blob));

ret = fdt\_shrink\_to\_minimum(blob, 0);

if (ret < 0)

goto err;

of\_size = ret;

if (\*initrd\_start && \*initrd\_end) {

of\_size += FDT\_RAMDISK\_OVERHEAD;

fdt\_set\_totalsize(blob, of\_size);

}

if (lmb)

lmb\_reserve(lmb, (ulong)blob, of\_size);

fdt\_initrd(blob, \*initrd\_start, \*initrd\_end);

if (!ft\_verify\_fdt(blob))

goto err;

return 0;

err:

printf(" - must RESET the board to recover.\n\n");

return ret;

}

检查dtb头部是否正确，然后从环境变量读出serial#的值，写入dtb的serial\_number字段即串口号。

fdt\_chosen

这个函数比较有趣，我们来分析一下：

看了这个函数是不是觉得有种很熟悉的感觉？ 对这个函数很重要的作用就是给kernel传递启动参数。在ARM64（不仅仅在K3）启动参数是写在dtb的chosen段。值就是bootargs。没有这个函数的处理，kernel是得不到启动参数的。

int fdt\_chosen(void \*fdt)

{

int nodeoffset;

int err;

char \*str; /\* used to set string properties \*/

err = fdt\_check\_header(fdt);

if (err < 0) {

printf("fdt\_chosen: %s\n", fdt\_strerror(err));

return err;

}

nodeoffset = fdt\_find\_or\_add\_subnode(fdt, 0, "chosen");

if (nodeoffset < 0)

return nodeoffset;

str = env\_get("bootargs");

if (str) {

err = fdt\_setprop(fdt, nodeoffset, "bootargs", str,

strlen(str) + 1);

if (err < 0) {

printf("WARNING: could not set bootargs %s.\n",

fdt\_strerror(err));

return err;

}

}

return fdt\_fixup\_stdout(fdt, nodeoffset);

}

Kernel在early\_init\_dt\_scan\_chosen这里取出启动参数。

arch\_fixup\_fdt

这个函数最重要的就是把memory的信息写入dtb，上面我们分析过K3有两个bank。一个是0x80000000开始的2G，给32bit应用使用。 另一个是0x880000000开始的2G，只供64bit。Linux能够使用的内存信息就是从这个函数给出的。

接下来的一些函数没有太多分析意义，除了fdt\_initrd

fdt\_initrd：把ramdisk的信息写入chosen字段，供Kernel启动后能够找到自己的rootfs。

好了，啰嗦了这么久，终于准备工作都完成了。接下来就是关闭MMU,ICACHE,DCACHE，这是linux的硬性要求，不然kernel起来后地址就全乱了。

armv8\_switch\_to\_el2((u64)images->ft\_addr, 0, 0, 0,

images->ep,

ES\_TO\_AARCH64);

X0即第一个参数，必须是dtb的地址。前面说过EL2级别是OS kernel运行的，所以arm64只需切换运行级别即可。

然后，u-boot就把控制权交给了linux kernel。一个伟大的系统就开始运行了。

啰啰嗦嗦写了这么多，我手上也没有K3的开发板，都是根据硬件手册+代码分析出来的，我知道里面肯定有很多错误，希望能给我斧正。