第 5 章

嵌入式操作系统初始化

大部分操作系统是由一个名为 BootLoader 的小程序调入内存并开始运行的。那么 BootLoader 引导程序是如何构造的呢？下面进行具体介绍。

5.1 BootLoader

简单地说，BootLoader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，可以初始化硬件设备，建立内存空间的映像图，从而使系统的软硬件环境工作在一个合适的状态，以便为最终调用操作系统内核做好准备。每种不同的 MCU 体系结构都有不同的 BootLoader 方式。有些 BootLoader 支持多种体系结构的 MCU，如 U-Boot 就同时支持 ARM 体系结构和 MIPS 体系结构。除了依赖于 MCU 的体系结构外，BootLoader 实际上也依赖于具体的嵌入式板级设备的配置。也就是说，对于两块不同的嵌入式板而言，即使它们是基于同一种 MCU 构建的，要想让运行在一块板子上的 BootLoader 程序也能运行在另一块板子上，通常也都需要修改 BootLoader 的源程序。

5.1.1 BootLoader 装在哪里

系统加电或复位后，所有的 MCU 通常都从由 MCU 制造商预先安排的某个地址上取指令。例如，基于 ARM7TDMI Core 的 MCU 在复位时通常都从地址 0x00000000 处取它的第一条指令。某种类型的物理存储设备（如 ROM、EEPROM 或 Flash 等）被映像到这个预先安排的地址上，BootLoader 程序就放在这里。

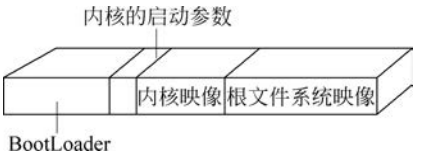

图 5.1 物理存储设备的典型空间分配结构

如图 5.1 所示就是一个同时装有 BootLoader、内核的启动参数、内核映像和根文件系统映像的物理存储设备的典型空间分配结构。

5.1.2 BootLoader 的启动过程

多阶段的 BootLoader 能提供更为复杂的功能，以及更好的可移植性。一般从存储设备

上启动的 BootLoader 大多都是两个阶段，即启动过程可以分为 stage1 和 stage2 两部分。stage1 中通常存放依赖于 MCU 体系结构的代码，如设备初始化代码等，通常都用汇编语言来实现，以达到短小、精悍的目的。stage2 的代码则通常用 C 语言来实现，这样可以实现更复杂的功能，而且代码会具有更好的可读性和可移植性。一个典型简单的 BootLoader 启动流程如图 5.2 所示。

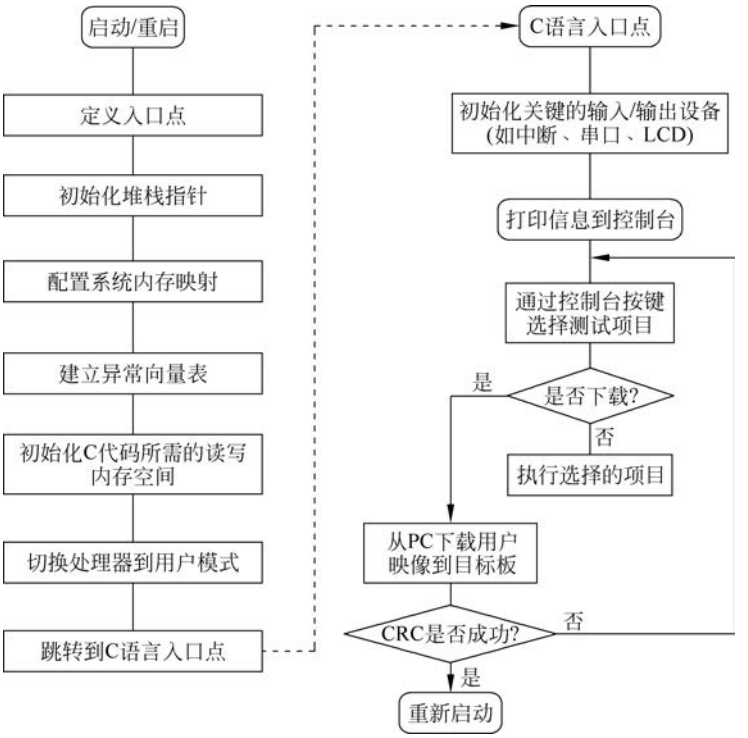

图 5.2　BootLoader 启动流程

1. stage1 中的初始化操作

BootLoader 的 stage1 主要包含依赖于 MCU 的体系结构硬件初始化的代码，通常都用汇编语言来实现。这个阶段的任务通常包括基本的硬件设备初始化(屏蔽所有的中断、关闭处理器内部指令/数据 cache 等)，为 stage2 准备 RAM 空间；如果是从某个固态存储到媒质中，则复制 BootLoader 的 stage2 代码到 RAM，然后设置堆栈，最后跳转到第二阶段的 C 程序入口点。

BootLoader 一开始就执行的这些操作，其目的是为 stage2 的执行以及随后的内核的执行准备好一些基本的硬件环境。它通常包括以下步骤(按执行的先后顺序)：

(1) 屏蔽所有的中断。在 BootLoader 的执行全过程中可以不必响应任何中断，以免去太庞大和复杂的设计。中断屏蔽可以通过写 MCU 的中断屏蔽寄存器或状态寄存器(如 ARM 的 CPSR 寄存器)来完成。

(2) 设置 MCU 的速度和时钟频率。有些 MCU 有多种速度模式,可工作在多个时钟频率下,此时需要选择其工作速度和时钟频率。

(3) RAM 初始化。这包括正确地设置系统的内存控制器的功能寄存器以及各内存控制寄存器等。

(4) 初始化 LED。一般是通过 GPIO 来驱动 LED 的。LED 不是系统必要的硬件配置,设置 LED 的目的是向系统安装调试人员表明系统的状态是正常(OK)的还是错误(ERROR)。如果板子上没有 LED,那么也可以通过初始化 UART 向串口打印 BootLoader 的 Logo 字符信息来完成这一点。

(5) 关闭 MCU 内部指令/数据 cache。通常使用 cache 以及写缓冲是为了提高系统性能,但由于 cache 的使用可能改变访问主存的数量、类型和时间,因此 BootLoader 通常是不需要 cache 工作的。

(6) 为加载 BootLoader 的 stage2 准备 RAM 空间。

为了获得更快的执行速度,通常把 stage2 加载到 RAM 空间中来执行,因此必须为加载 BootLoader 的 stage2 准备好一段可用的 RAM 空间范围。

由于 stage2 通常是 C 语言执行代码,因此在考虑空间大小时,除了首先考虑 stage2 可执行映像的大小外,还必须把堆栈空间也考虑进来。其次,空间大小最好是内存页(memory page)大小(通常是 4KB)的倍数。一般而言,1MB 的 RAM 空间已经足够了。具体的地址范围可以任意安排,如 BLOB(即 BootLoader Object,是一款功能强大的 BootLoader。它遵循 GPL,源代码完全开放。BLOB 既可以用来简单调试,也可以启动 Linux 内核)就将它的 stage2 可执行映像安排到从系统 RAM 起始地址 0xc0200000 开始的 1MB 空间内执行。但是,将 stage2 安排到整个 RAM 空间的最顶端那个 1MB 空间,即(RamEnd－1MB)～RamEnd,是一种很好的设计方法。

为了后面的叙述方便,这里把所安排的 RAM 空间范围的大小记为 stage2_size(字节)。把起始地址和终止地址分别记为 stage2_start 和 stage2_end(这两个地址均以 4 字节边界对齐)。因此:

```
stage2_end = stage2_start + stage2_size
```

另外,还必须确保所安排的地址范围是可读/写的 RAM 空间,为此,必须对所安排的地址范围进行测试。具体的测试方法可以采用类似于 blob 的方法,即以内存页为被测试单位,测试每个内存页开始的两个字是否是可读/写的。记这个检测算法为 test_mempage,其具体步骤如下:

① 先保存内存页一开始两个字的内容。

② 向这两个字中写入任意的数字。例如,向第一个字写入 0x55,向第二个字写入 0xaa。

③ 立即将这两个字的内容读回。显然,读到的内容应该分别是 0x55 和 0xaa。如果不是,则说明这个内存页所占据的地址范围不是一段有效的 RAM 空间。

④ 再向这两个字中写入任意的数字。例如,向第一个字写入 0xaa,向第二个字写入 0x55。

⑤ 立即将这两个字的内容读回。显然，读到的内容应该分别是 0xaa 和 0x55。如果不是，则说明这个内存页所占据的地址范围不是一段有效的 RAM 空间。

⑥ 恢复这两个字的原始内容。测试完毕。

⑦ 为了得到一段干净的 RAM 空间范围，也可以将所安排的 RAM 空间范围进行清零操作。

代码如下：

```
testram:
    stmdb  sp!, {r1 - r4, lr}
        ldmia  r0, {r1, r2}
        mov    r3, #0x55    @ write 0x55 to first word
        mov    r4, #0xaa    @ 0xaa to second
        stmia  r0, {r3, r4}
        ldmia  r0, {r3, r4} @ read it back
        teq    r3, #0x55    @ do the values match
        teqeq  r4, #0xaa
        bne    bad          @ oops, no
        mov    r3, #0xaa    @ write 0xaa to first word
        mov    r4, #0x55    @ 0x55 to second
        stmia  r0, {r3, r4}
        ldmia  r0, {r3, r4} @ read it back
        teq    r3, #0xaa    @ do the values match
        teqeq  r4, #0x55
bad:    stmia  r0, {r1, r2} @ in any case, restore old data
        moveq  r0, #0       @ ok - all values matched
        movne  r0, #1       @ no ram at this location
ldmia   sp!, {r1 - r4, pc}
```

(7) 复制 BootLoader 的 stage2 到 RAM 空间中。

复制时要先确定两点：

① stage2 的可执行映像在固态存储设备的存放起始地址和终止地址。

② RAM 空间的起始地址。

(8) 设置好堆栈。

堆栈指针的设置是为了执行 C 语言代码做好准备。通常可以把 sp 的值设置为：stage2_end－4，即在安排的那个 1MB 的 RAM 空间的最顶端(堆栈向下生长)。

此外，在设置堆栈指针 sp 之前，也可以关闭 LED 灯，以提示用户准备跳转到 stage2。经过上述这些执行步骤后，系统的物理内存布局应该如图 5.3 所示。

(9) 跳转到 stage2 的 C 语言入口点。

在上述一切都就绪后，就可以跳转到 BootLoader 的 stage2 去执行了。例如，可以通过修改 PC 寄存器为合适的地址来实现跳转。

2. BootLoader 的 stage2

stage2 主要包括提供灵活的程序入口、初始化本阶段要用到的硬件设备、检测系统的内存映像、加载内核映像和根文件系统映像、设置内核的启动参数和调用内核等。

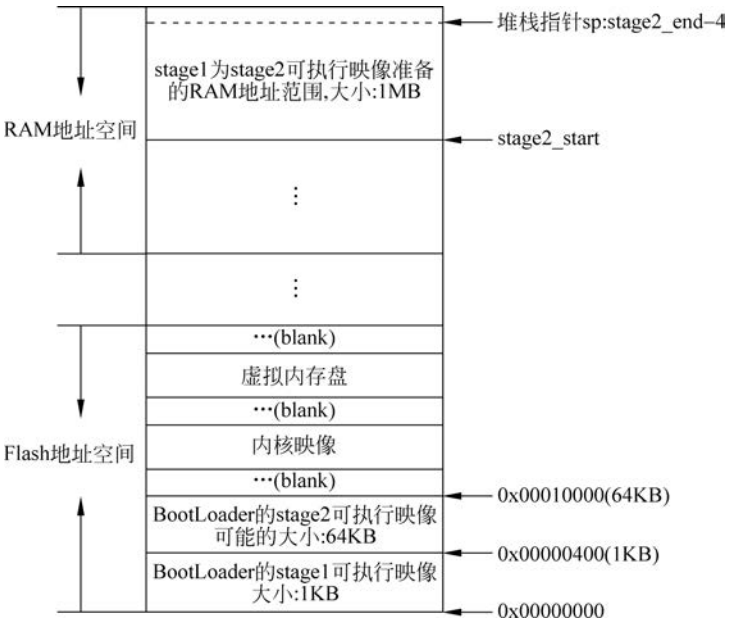

图 5.3 BootLoader 的 stage2 可执行映像刚被复制到 RAM 空间时的系统物理内存布局

(1) 提供灵活的程序入口。

正如前面所说,stage2 的代码通常用 C 语言来实现,以便于实现更复杂的功能和取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是:在编译和链接 BootLoader 这样的初始程序时,不能使用 glibc 库中的任何支持函数。这就带来一个问题,那就是从哪里跳转进 main()函数呢?直接把 main()函数的起始地址作为整个 stage2 执行映像的入口点或许是最直接的想法。但是这样做有两个缺点:第一,无法通过 main()函数传递函数参数;第二,无法处理 main()函数返回的情况。更为巧妙的方法是利用 trampoline(弹簧床)的概念,用这段 trampoline 小程序作为 main()函数的外部包裹(external wrapper)。也就是说,用汇编语言写一段 trampoline 小程序,并将这段 trampoline 小程序作为 stage2 可执行映像的执行入口点,然后可以在 trampoline 汇编小程序中用 MCU 跳转指令跳入 main()函数中去执行,而当 main()函数返回时,MCU 执行路径显然再次回到 trampoline 程序。

下面给出一个简单的 trampoline 程序示例。

```
.text
.globl _trampoline
_trampoline:
    bl  main
    b   _trampoline
```

可以看出，当 main()函数返回后，又用一条跳转指令重新执行 trampoline 程序——当然也就重新执行 main()函数，这也就是 trampoline(弹簧床)一词的意思所在。

(2) 初始化本阶段要用到的硬件设备。

这通常包括：初始化至少一个串口，以便和终端用户进行信息输出；初始化计时器等。在初始化这些设备之前，也可以重新把 LED 灯点亮，以表明已经进入 main()函数执行。设备初始化完成后，可以输出一些打印信息，如程序名字字符串、版本号等。

(3) 检测系统的内存映像。

所谓内存映像就是指在整个 4GB 物理地址空间中有哪些地址范围被分配用来寻址系统的 RAM 单元。例如，在 SA-1100 MCU 中，从 0xc0000000 开始的 512MB 地址空间被用作系统的 RAM 地址空间；而在 Samsung S3C44B0X MCU 中，从 0x0c000000～0x10000000 的 64MB 地址空间被用作系统的 RAM 地址空间。虽然 MCU 通常预留出一大段足够的地址空间给系统 RAM，但是在搭建具体的嵌入式系统时却不一定会实现 MCU 预留的全部 RAM 地址空间。也就是说，具体的嵌入式系统往往只把 MCU 预留的全部 RAM 地址空间中的一部分映像到 RAM 单元上，而让剩下的那部分预留 RAM 地址空间处于未使用状态。鉴于上述这个事实，BootLoader 的 stage2 必须在它想要工作(如将存储在 Flash 上的内核映像读到 RAM 空间中)之前检测整个系统的内存映像情况，即它必须知道 MCU 预留的全部 RAM 地址空间中的哪些被真正映像到 RAM 地址单元，哪些是处于 unused 状态的。

① 内存映像的描述。可以用如下数据结构来描述 RAM 地址空间中一段连续(continuous)的地址范围：

```
typedef struct memort_area_struct{
u32 start; /* 内存区域基地址 */
u32 size;  /* 内存区域字节数 */
int used;
} memory_area_t;
```

这段 RAM 地址空间中的连续地址范围可以处于两种状态之一：如果 used=1，则说明这段连续的地址范围已被实现，即真正地被映像到 RAM 单元上；如果 used=0，则说明这段连续的地址范围并未被系统所实现，而是处于未使用状态。

基于上述 memory_area_t 数据结构，整个 MCU 预留的 RAM 地址空间可以用一个 memory_area_t 类型的数组来表示，代码如下：

```
memory_area_t memory_map[NUM_MEM_AREAS] = {
[0 ... (NUM_MEM_AREAS - 1)] = {
    .start = 0,
    .size = 0,
    .used = 0
  }
}
```

② 内存映像的检测。下面给出一个可用来检测整个 RAM 地址空间内存映像情况的

简单而有效的算法：

```
/* 数组初始化 */
for(i = 0; i < NUM_MEM_AREAS; i++)
    memory_map[i].used = 0;
    /* first write a 0 to all memory locations */
for(addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE)
    * (u32 *)addr = 0;
for(i = 0, addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE)
 {
    /*
    * 检测从基地址 MEM_START + i * PAGE_SIZE 开始,大小为
    * PAGE_SIZE 的地址空间是否为有效的 RAM 地址空间
    */
    调用前面提到的算法 test_mempage();
    if ( current memory page isnot a valid ram page)
     {
        /* 不是 RAM */
        if(memory_map[i].used )
            i++;
        continue;
    }
    /*
    * 当前页已经是一个被映像到 RAM 的有效地址范围

    * 但是还要确定当前页是否只是 4GB 地址空间中某个地址页的别名
    */
    if( * (u32 *)addr != 0)
     { /* 有别名吗? */
        /* 这个内存页是 4GB 地址空间中某个地址页的别名 */
        if ( memory_map[i].used )
            i++;
        continue;
    }
    /*
    * 当前页已经是一个被映像到 RAM 的有效地址范围
    * 而且它也不是 4GB 地址空间中某个地址页的别名
    */
    if (memory_map[i].used == 0)
    {
        memory_map[i].start = addr;
        memory_map[i].size = PAGE_SIZE;
        memory_map[i].used = 1;
    }
     else
     {
        memory_map[i].size += PAGE_SIZE;
    }
} /* for 循环结束 */
```

在用上述算法检测完系统的内存映像情况后，BootLoader 也可以将内存映像的详细信息打印到串口。

(4) 加载内核映像和根文件系统映像。

① 规划内存占用的布局。这里包括两个方面的规划：第一，内核映像所占用的内存范围；第二，根文件系统映像所占用的内存范围。在规划内存占用的布局时，主要考虑基地址和映像的大小两个方面。

对于内核映像，一般将其复制到从 MEM_START＋0x8000 这个基地址开始的大约 1MB 的内存范围内(嵌入式 Linux 的内核一般都不超过 1MB)。为什么要把从 MEM_START～MEM_START＋0x8000 这段 32KB 大小的内存空出来呢？这是因为 Linux 内核要在这段内存中放置一些全局数据结构，如启动参数和内核页表等信息。

对于根文件系统映像，则一般将其复制到 MEM_START＋0x00100000 开始的地方。如果用 ramdisk 作为根文件系统映像，则其解压后的大小一般是 1MB。

② 从 Flash 上复制。一般嵌入式 MCU 通常都是在统一的内存地址空间中寻址 Flash 等固态存储设备的，因此从 Flash 上读取数据与从 RAM 单元中读取数据并没有什么不同。用一个简单的循环就可以完成从 Flash 设备上复制映像的工作：

```
while(count) {
    *dest++ = *src++;  /* 所有的都以字为单位对齐 */
    count -= 4;        /* 字节数 */
};
```

(5) 设置内核的启动参数。

应该说，在将内核映像和根文件系统映像复制到 RAM 空间中后，就可以准备启动操作系统内核了。但是在调用内核之前，应该做一步准备工作，即设置操作系统内核的启动参数。

这里以 Linux 为例，Linux 2.4.x 以后的内核都期望以标记列表(tagged list)的形式来传递启动参数。启动参数标记列表以标记 ATAG_CORE 开始，以标记 ATAG_NONE 结束。每个标记由标识被传递参数的 tag_header 结构以及随后的参数值数据结构组成。数据结构 tag 和 tag_header 定义在 Linux 内核源码的 include/asm/setup.h 头文件中。

```
/* 列表以 ATAG_NONE 节点为结尾 */
#define ATAG_NONE  0x00000000
struct tag_header
{
    u32 size; /* 注意，这里 size 是以字节数为单位的 */
    u32 tag;
};
……
struct tag
{
    struct tag_header hdr;
```

```
    union
    {
        struct tag_core       core;
        struct tag_mem32      mem;
        struct tag_videotext  videotext;
        struct tag_ramdisk    ramdisk;
        struct tag_initrd     initrd;
        struct tag_serialnr   serialnr;
        struct tag_revision   revision;
        struct tag_videolfb   videolfb;
        struct tag_cmdline    cmdline;
        /* Acorn 结构体声明 */
        struct tag_acorn      acorn;
        /* DC21285 定义声明 */
        struct tag_memclk     memclk;
    } u;
};
```

在嵌入式 Linux 系统中，通常需要由 BootLoader 设置启动参数，包括 ATAG_CORE、ATAG_MEM、ATAG_CMDLINE、ATAG_RAMDISK、ATAG_INITRD 等。

例如，设置 ATAG_CORE 的代码如下：

```
params = (struct tag *)BOOT_PARAMS;
params->hdr.tag = ATAG_CORE;
params->hdr.size = tag_size(tag_core);
params->u.core.flags = 0;
params->u.core.pagesize = 0;
params->u.core.rootdev = 0;
params = tag_next(params);
```

其中，BOOT_PARAMS 表示内核启动参数在内存中的起始基地址；指针 params 是一个 struct tag 类型的指针；宏 tag_next()将以指向当前标记的指针为参数，计算紧邻当前标记的下一个标记的起始地址。注意，内核的根文件系统所在的设备 ID 就是在这里设置的。

下面是设置内存映像的示例代码：

```
for(i = 0; i < NUM_MEM_AREAS; i++) {
  if(memory_map[i].used) {
      params->hdr.tag = ATAG_MEM;
      params->hdr.size = tag_size(tag_mem32);
      params->u.mem.start = memory_map[i].start;
      params->u.mem.size = memory_map[i].size;
      params = tag_next(params);
  }
}
```

可以看出，在 memory_map[]数组中，每个有效的内存段都对应一个 ATAG_MEM 参

数标记。

Linux内核在启动时可以命令行参数的形式来接收信息，利用这一点可以向内核提供那些内核不能自己检测的硬件参数信息，或者重载(override)内核自己检测到的信息。例如用这样一个命令行参数字符串 console=ttyS0,115200n8 来通知内核以 ttyS0 作为控制台，且串口采用“115200b/s、无奇偶校验、8位数据位”的设置。下面是一段设置调用内核命令行参数字符串的示例代码：

```
char * p;
/* eat leading white space */
for(p = commandline; *p == ''; p++)
    /*跳过不存在的命令行,所以内核会停在默认命令行。*/
    if(*p == '\0')
        return;
params->hdr.tag = ATAG_CMDLINE;
params->hdr.size = (sizeof(struct tag_header) + strlen(p) + 1 + 4) >> 2;
strcpy(params->u.cmdline.cmdline, p);
params = tag_next(params);
```

注意在上述代码中设置 tag_header 的大小时，必须包括字符串的终止符'\0'，此外还要将字节数向上调整4字节，因为 tag_header 结构中的 size 成员表示的是字数。

下面是设置 ATAG_INITRD 的示例代码，它指出内核在 RAM 中的什么地方可以找到 initrd 映像(压缩格式)以及它的大小。

```
params->hdr.tag = ATAG_INITRD2;
params->hdr.size = tag_size(tag_initrd);
params->u.initrd.start = RAMDISK_RAM_BASE;
params->u.initrd.size = INITRD_LEN;
params = tag_next(params);
```

下面是设置 ATAG_RAMDISK 的示例代码，它指出内核解压后的 ramdisk 有多大(单位是 KB)。

```
params->hdr.tag = ATAG_RAMDISK;
params->hdr.size = tag_size(tag_ramdisk);
params->u.ramdisk.start = 0;
params->u.ramdisk.size = RAMDISK_SIZE;  /* 注意,单位是 KB */
params->u.ramdisk.flags = 1;            /* 自动加载 ramdisk */
params = tag_next(params);
```

最后，设置 ATAG_NONE 标记，结束整个启动参数列表。

```
static void setup_end_tag(void)
{
    params->hdr.tag = ATAG_NONE;
    params->hdr.size = 0;
}
```

(6) 调用内核。

BootLoader 调用 Linux 内核的方法是直接跳转到内核的第一条指令处,即直接跳转到 MEM_START+0x8000 地址处。在跳转时,要满足下列条件:

① MCU 寄存器的设置。

R0=0。

R1=机器类型 ID。

关于机器类型编号(Machine Type Number),可以参见 Linux/arch/arm/tools/mach-types。

R2=启动参数标记列表在 RAM 中的起始基地址。

② MCU 模式。

必须禁止中断 IRQs 和 FIQs。

MCU 必须为 SVC 模式。

③ cache 和 MMU 的设置。

MMU 必须关闭。

指令 cache 可以打开,也可以关闭。

数据 cache 必须关闭。

如果用 C 语言,可以像下列示例代码这样来调用内核:

```
void ( * theKernel)(int zero, int arch, u32 params_addr) = (void ( * )(int,
int, u32))KERNEL_RAM_BASE;
……
theKernel(0, ARCH_NUMBER, (u32) kernel_params_start);
```

注意,theKernel()函数调用应该永远不返回。如果这个调用返回,则说明出错。

5.1.3 基于 MicroBlaze 软核处理器的 BootLoader 设计

FPGA(Field Programmable Gate Array,现场可编程逻辑门阵列)必须先将内部硬件逻辑配置完成之后,才能运行程序代码。虽然可以直接将程序代码例化到片内 BRAM(Buffer Random Access Memery,缓存区随机存取器)中,但是由于 FPGA 内部的 BRAM 资源有限,而且硬件逻辑配置时就会占用其中的资源,因此遇到大型系统设计时(如带有 TCP/IP 的大型程序),就必须使用外部的 RAM 来存储程序代码和堆栈,这就需要设计 BootLoader 程序来完成用户程序的引导。BootLoader 程序是在 FPGA 硬件配置完毕之后,在内部处理器上运行的一段启动代码,用来将 Flash 中的用户程序传输至外部 RAM,并引导嵌入式系统从用户程序中开始运行,EDK(Embedded Development Kit,嵌入式开发套件)根据用户选用 IP 核搭建出系统结构,生成 MHS(Microprocessor Hardware Specification,微处理器硬件规范)文件。该文件中主要定义了系统硬件细节、MicroBlaze 软核、SPI 控制 IP 核等的具体配置参数、系统所需的各种存储空间的地址分配。MHS 文件生成后,EDK 根据该文件以及 FPGA 的其余功能文件综合生成下载配置文件,此时硬件设计部分完成。

软件部分的设计应包括 BootLoader 程序设计、系统及用户程序设置、配置文件制作 3 个部分。

BootLoader 程序主要由驱动程序和应用程序两个部分组成。系统加电或复位后，处理器通常都从预先确定的地址上取指令。一般来说，处理器复位时都从地址 0x00000000 处取它的第一条指令。而嵌入式系统通常都有某种类型的固态存储设备（如 ROM、EEPROM 或 Flash 等）被安排在这个起始地址上。但是对于 MicroBlaze 软核处理器，起始地址分配给了可引导的挂载在 PLB（Processor Local Bus，处理器内部总线）总线上的 BRAM 存储器（该存储器是使用 FPGA 的内部资源例化而成的）。通过开发工具 EDK 可以将 BootLoader 程序定位在起始地址开始的存储空间内。所以，BootLoader 程序是系统加电后、用户应用程序运行之前必须运行的一段引导代码。具体包括：①硬件设备初始化；②复制用户程序到 RAM 空间（DDR SDRAM）；③校验已复制的用户程序；④指针跳转到预先设定的用户程序 RAM 空间首地址。

FPGA 中做 BootLoader 的软件流程如图 5.4 所示。

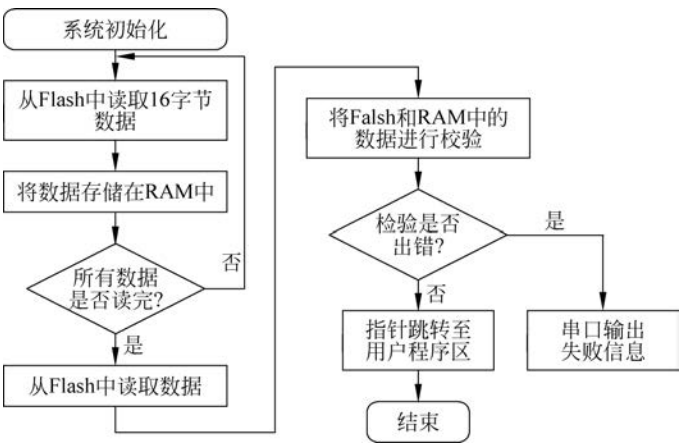

图 5.4　FPGA 中做 BootLoader 的软件流程

对于系统及用户程序设置过程，首先需要将 BootLoader 工程例化到 BRAMs 中，即系统会先运行 BootLoader 程序。其次，修改应用程序存放的地址空间，即将应用程序写入 DDR SDRAM 的首地址。同时，修改配置文件参数，将引导时钟源由 JTAG（Joint Test Action Group，联合测试工作组）的时钟引脚修改为 FPGA 的 CCLK 引脚。FPGA 通过它的 CCLK 引脚输出时钟信号，引导整个配置过程。

在配置文件制作过程中，首先用 Xilinx 公司的 ISE 软件的 iMPACT 工具将原系统配置文件转换为可下载至 Flash 的 MCS（Modulation and Coding Scheme，编码调制方案）文件。其次，将系统编译用户程序生成的 ELF（Executable and Linkable Format，可执行连接文件格式）文件转换为可下载的 MCS 文件，并将第一次生成的 MCS 文件和刚才生成的 MCS 文件合并为最终的配置文件。最后，使用 iMPACT 工具通过 JTAG 口，将配置文件下载至

Flash 中，此时整个系统才构建完毕。

5.1.4 基于 STM32 处理器的简单 BootLoader 设计

BootLoader 其实就是一段启动程序，它在芯片启动时首先被执行，它可以用来做一些硬件的初始化，当初始化完成之后跳转到对应的应用程序中。

首先，将内存分为两个区：一个是启动程序区(0x0800 0000～0x0800 2000)，大小为8KB；另一个为应用程序区(0x0800 2000～0x0801 0000)。

其次，芯片上电时先运行启动程序，然后跳转到应用程序区执行应用程序。

1. 跳转实现

BootLoader 一个主要的功能就是程序的跳转。在 STM32 中只要将要跳转的地址直接写入 PC 寄存器，就可以跳转到对应的地址中。

当实现一个函数时，这个函数最终会占用一段内存，而它的函数名代表的就是这段内存的起始地址。当调用这个函数时，单片机会将这段内存的首地址(函数名对应的地址)加载到 PC 寄存器中，从而跳转到这段代码来执行。那么也可以利用这个原理定义一个函数指针，将这个指针指向想要跳转的地址，然后调用这个函数，就可以实现程序的跳转了。

跳转程序设计如下：

```
#define APP_ADDR 0x08002000                 //应用程序首地址定义
typedef void ( * APP_FUNC)();               //函数指针类型定义

APP_FUNC jump2app;                          //定义一个函数指针

jump2app = ( APP_FUNC )(APP_ADDR + 4);      //给函数指针赋值
jump2app();                                 //调用函数指针,实现程序跳转
```

上面的代码实现了跳转功能，但是为什么要跳转到(APP_ADDR+4)这个地址，而不是APP_ADDR 呢？

首先要了解主控芯片的启动过程。以 STM32 为例，在芯片上电时，先从内存地址位0x08000000(由启动模式决定)的地方加载栈顶地址(4B)，再从 0x08000004 的地方加载程序复位地址(4B)，然后跳转到对应的复位地址去执行。

所以上面的程序中，jump2app 这个函数指针的地址为(APP_ADDR+4)，调用这个函数指针时，芯片内核会自动跳转到这个指针指向的内存地址，即应用程序的复位地址。

2. 栈地址加载

根据前面讲的 STM32 的硬件知识，程序需要切换，就需要找到程序所用的堆栈，让寄存器指向堆栈。为了能够在启动时找到栈，完整的栈地址加载和跳转程序如下：

```
__ asm void MSR_MSP(uint32_t addr)
{
    MSR MSP, r0
    BX r14;
```

```
}
```

__asm void MSR_MSP(uint32_t addr) 是 MDK 嵌入式汇编形式。

MSR MSP,r0 的意思是将 r0 寄存器中的值加载到 MSP(主栈寄存器,复位时默认使用)寄存器中,r0 中保存的是参数值,即 addr 的值。

BX r14 跳转到连接寄存器保存的地址中,即退出函数,跳转到函数调用地址。

完整的程序如下:

```
#define APP_ADDR 0x08002000                    //应用程序首地址定义
typedef void (*APP_FUNC)();                    //函数指针类型定义

/**
  * @brief
  * @param
  * @retval
  */
__asm void MSR_MSP(uint32_t addr)
{
    MSR MSP, r0
    BX r14;
}


/**
  * @brief
  * @param
  * @retval
  */
void run_app(uint32_t app_addr)
{
    uint32_t reset_addr = 0;
    APP_FUNC jump2app;

    /* 跳转之前关闭相应的中断 */
    NVIC_DisableIRQ(SysTick_IRQn);
    NVIC_DisableIRQ(LPUART_IRQ);

    /* 栈顶地址是否合法(这里 SRAM 大小为 8KB) */
    if((*(uint32_t *)app_addr)&0x2FFFE000) == 0x20000000)
    {
        /* 设置栈指针 */
        MSR_MSP(app_addr);
        /* 获取复位地址 */
        reset_addr = *(uint32_t *)(app_addr+4);
        jump2app = (APP_FUNC)reset_addr;
        jump2app();
```

```
    }
    else
    {
        printf("APP Not Found!\n");
    }
}
```

3. 编译设置

任何嵌入式程序要运行时都需要在编译器中设置程序的存储地址。如图 5.5 所示，这个程序按照上面讨论的需要在目标(target)设置界面将默认(0x8000000)改为应用程序地址(0x8002000)。

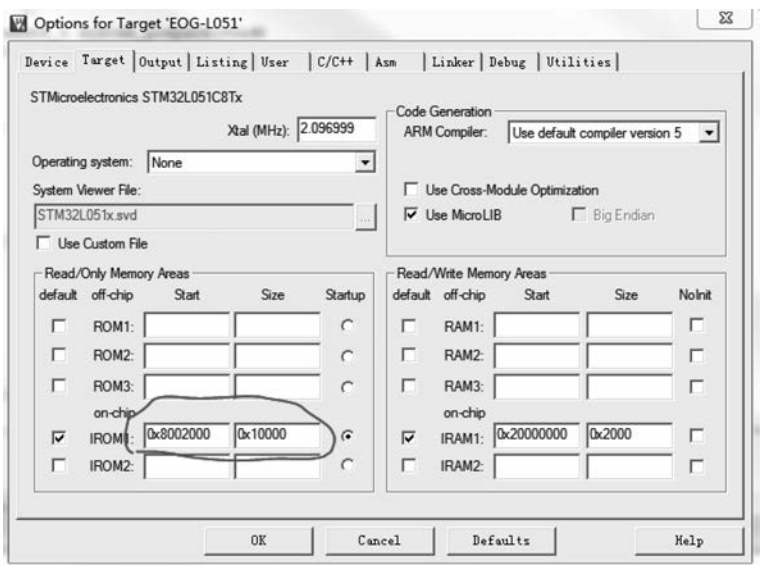

图 5.5　在编译器中修改应用程序地址

4. 中断向量表重映射

系统原有的.s 文件中有如下代码：

```
Reset handler routine
Reset_Handler PROC
            EXPORT Reset_Handler        [WEAK]
      IMPORT __main
      IMPORT SystemInit
            LDR     R0, =SystemInit
            BLX     R0
            LDR     R0, =__main
            BX      R0
            ENDP
```

这段代码表示，程序在执行 main 函数之前，会先执行 SystemInit 这个函数。这个函数

主要是做了时钟的初始化和中断初始化，还有就是中断向量表的映射（就是最后那一段代码）。整个函数如下：

```
/**
  * @brief Setup the microcontroller system.
  * @param None
  * @retval None
  */
void SystemInit (void)
{
/* 设置 MSION */
  RCC->CR |= (uint32_t)0x00000100U;

  /* 重置 SW[1:0],HPRE[3:0],PPRE1[2:0],PPRE2[2:0],MCOSEL[2:0]和 MCOPRE[2:0]位 */
  RCC->CFGR &= (uint32_t) 0x88FF400CU;

  /* 重置 HSION,HSIDIVEN,HSEON,CSSON 和 PLLON 位 */
  RCC->CR &= (uint32_t)0xFEF6FFF6U;

  /* 重置 HSI48ON */
  RCC->CRRCR &= (uint32_t)0xFFFFFFFEU;

  /* 重置 HSEBYP */
  RCC->CR &= (uint32_t)0xFFFBFFFFU;

  /* 重置 PLLSRC,PLLMUL[3:0]和 PLLDIV[1:0] */
  RCC->CFGR &= (uint32_t)0xFF02FFFFU;

  /* 失效所有中断 */
  RCC->CIER = 0x00000000U;

  /* 向量表重映射 */
#ifdef VECT_TAB_SRAM
  SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal SRAM */
#else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal FLASH */
#endif
}
```

将其改成 SCB-> VTOR＝0x8002000 就可以了。

其他更多的功能读者自行添加。

第 19 集
视频讲解

5.2　嵌入式操作系统初始化数据结构及主要操作

第 20 集
视频讲解

当嵌入式操作系统微内核被调入后，系统开始进行初始化。所谓初始化，就是在内存中开始生成一些数据结构，支持后续的操作。

5.2.1 μCOS-Ⅱ主要数据结构及操作

μCOS-Ⅱ初始化后，系统的主要数据结构包括5个链表控制块和数组、位图等。

如图4.2所示是μCOS-Ⅱ初始化的数据结构。左边有一个任务就绪组和一个任务就绪表相配合。任务就绪组是一个8位的变量；任务就绪表是一个位图，也就是一个有8个元素的数组，每个元素都是8位。任务就绪组的每一位和任务就绪表的一行相对应，像是给任务就绪表建立的索引，表示任务就绪表此行是否有1的元素。如果此行有一个1，任务就绪组的这一位就置1；如果任务就绪表此行全部为0，任务就绪组就清为0。

1. 任务就绪组(OSRdyGrp)和任务就绪表(OSRdyTbl[])

任务就绪组和任务就绪表一起用来帮助系统快速查到最高优先级的任务。因此，任务创建、删除等都会对任务就绪组和任务就绪表产生影响。

2. 优先级映像表(OSMapTbl[])

为了能够快速设置得到当前任务，免去复杂计算，特别增加了优先级映像表。

优先级映像表OSMapTbl[]（见图5.6）中，每个优先级所对应的高三位和低三位依次对应一个二进制值，查到OSMapTbl[]对应的值后可以很容易地置OSRdyGrp和OSRdyTbl[]对应位为1。

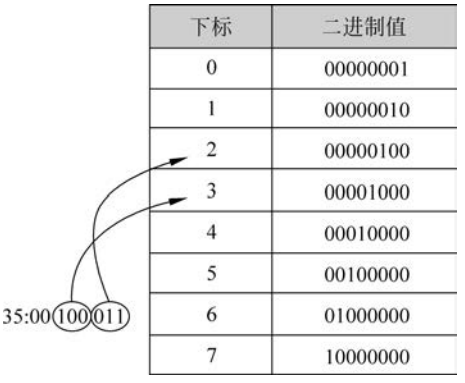

图5.6 优先级映像表OSMapTbl[]

优先级映像表char OSMapTbl[8]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80}，每个元素中只有一个位为1，这个1的位置暗含了优先级的位置。例如优先级35，用二进制写成00100011，按从低到高三位取数，低三位是011，对应十进制值3，那么直接查下标为2（数组从0开始）的元素，值为00000100，1的位置恰好是第三位。而接下来的三位是100，对应十进制值4，查下标可得OSMapTbl[3]=00001000，1的位置恰好是第四位。这样利用一张表，直接可以查出1的位置，从而省去了每次在程序中计算的工作量，是一种以空间换时间的方法。

3. 优先级决策表(OSUnMapTbl[])

优先级决策表是用来查当前系统中哪个优先级最高的一个矩阵。例如，当前系统的优先级就绪组里存放的是0x68，对应的优先级就绪表里存放的是0xE4，那么通过查表得OSUnMapTbl[0x68]和OSUnMapTbl[0xE4]的值分别是3和2，那么把3(011)作为高位，2(010)作为低位，得到的011010的值就是26。也就是说，当前系统中最高的优先级是26，如图5.7所示。

4. 任务控制优先级映像表

任务控制优先级映像表是一个有64个元素的数组，对应64个任务优先级，分别用来存放已分配过的任务控制块的指针，方便在运行时快速获取该任务的控制块指针。

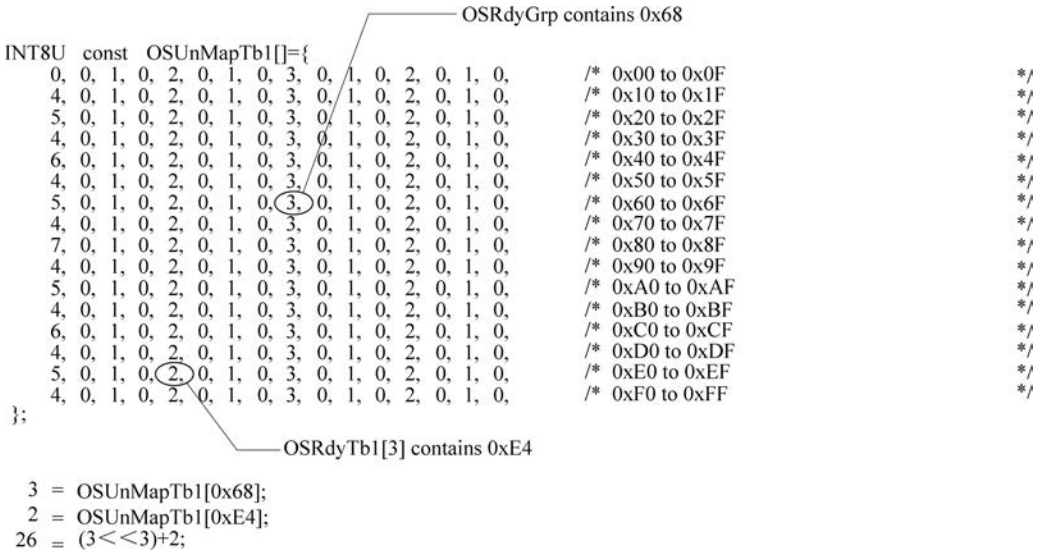

图 5.7　优先级决策表 OSUnMapTbl[]

5. 任务控制块

任务控制块是一个结构体数据结构，用于记录各个任务的信息。当任务的 MCU 的使用权被剥夺时，μCOS-Ⅱ用它来保存任务的当前状态；当任务重新获得 MCU 的使用权时，任务控制块能确保任务从当时被中断的那一点丝毫不差地继续执行。任务控制块全部存放在 RAM 中。

任务块是如下的数据结构：

```
typedef struct os_tcb {
OS_STK * OSTCBStkPtr;
/* 指向当前任务使用的堆栈的栈顶。μCOS-Ⅱ允许每个任务堆栈的大小不同,这样用户可以根据
实际需要定义任务堆栈的大小,可以节省 RAM 的空间。另外,由于 OSTCBStkPtr 是该结构体中的第一
个变量,所以可以使用汇编语言方便地访问,因为其偏移量是 0。当切换任务时,用户可以容易地知
道就绪任务中优先级最高任务的栈顶 */
#if OS_TASK_CREATE_EXT_EN > 0
void * OSTCBExtPtr;      /* 指向用户定义的扩展任务控制块 */
OS_STK * OSTCBStkBottom;
/* 指向任务堆栈的栈底。需要考虑一下使用的 MCU 的栈指针是按照从高
到低还是从低到高变化的。这个变量在测试任务需要的栈空间时需要使用 */
INT32U OSTCBStkSize;     /* 同样,该变量也是在测试任务需要的栈空间时需要用到的。需要注意
                            的是,该变量存储的是指针元的数目,而不是字节数目 */
INT16U OSTCBOpt;         /* 传给函数 OSTaskCreateExt()的选择项。目前有 OS_TASK_OPT_STK_CHK、
                            OS_TASK_OPT_STK_CLR 和 OS_TASK_OPT_SAVE_EP */
INT16U OSTCBId;          /* Task ID (0..65535) */
#endif
```

```
struct os_tcb  * OSTCBNext;
struct os_tcb  * OSTCBPrev;
/* 指向 TCB 的双向链表的前后链接,在 OSTimeTick()中使用,用来刷新各任务的任务延迟变量
OSTCBDly */
#if (OS_EVENT_EN) || (OS_FLAG_EN > 0u)
OS_EVENT  * OSTCBEventPtr;                    /* 指向事件控制块的指针 */
#endif
#if (OS_EVENT_EN) && (OS_EVENT_MULTI_EN > 0)
OS_EVENT  ** OSTCBEventMultiPtr;              /* 指向多重事件控制块的指针 */
#endif
#if ((OS_Q_EN > 0u) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0u)
void  * OSTCBMsg;                             /* 指向传递给任务的消息的指针 */
#endif
#if (OS_FLAG_EN > 0u) && (OS_MAX_FLAGS > 0)
#if OS_TASK_DEL_EN > 0
OS_FLAG_NODE  * OSTCBFlagNode;                /* 指向事件标志的节点的指针 */
#endif
OS_FLAGS OSTCBFlagsRdy; /* 当任务等待事件标志组时,该变量是使任务进入就绪态的事件标志 */
#endif
INT32U OSTCBDly;                              /* 记录事件延时或者挂起的时间 */
INT8U OSTCBStat;                              /* 任务状态字,如就绪状态、等待 */
INT8U OSTCBStatPend;                          /* 任务挂起状态 */
INT8U OSTCBPrio;                              /* 任务优先级  */
INT8U OSTCBX;                                 /* 计算优先级用 */
INT8U OSTCBY;                                 /* 计算优先级用 */
#if OS_LOWEST_PRIO <= 63
INT8U OSTCBBitX;                              /* 计算优先级用 */
INT8U OSTCBBitY;                              /* 计算优先级用 */
#else
INT16U OSTCBBitX;                             /* 计算优先级用 */
INT16U OSTCBBitY;                             /* 计算优先级用 */
#endif
#if OS_TASK_DEL_EN > 0
INT8U OSTCBDelReq;                            /* 表示任务是否需要删除 */
#endif
#if OS_TASK_PROFILE_EN > 0
INT32U OSTCBCtxSwCtr;                         /* 任务切换的次数 */
INT32U OSTCBCyclesTot;                        /* 任务运行的时钟周期数 */
INT32U OSTCBCyclesStart;                      /* 任务恢复开始的循环计数器 */
OS_STK  * OSTCBStkBase;                       /* 指向任务栈开始的指针 */
INT32U OSTCBStkUsed;                          /* 使用的栈的字节数  */
#endif
#if OS_TASK_NAME_EN > 0
INT8U  * OSTCBTaskName;                       /* 任务名称  */
#endif
#if OS_TASK_REG_TBL_SIZE > 0
INT32U OSTCBRegTbl[OS_TASK_REG_TBL_SIZE];     /* 任务注册表  */
```

```
#endif
} OS_TCB;
```

6. 任务就绪表和任务就绪组针对各种任务操作的变化

(1) 任务产生/任务进入就绪。

任务产生时,会根据情况分配优先级,μCOS-Ⅱ中的任务是靠优先级进行识别的。某个优先级任务产生时,首先需要将该优先级插入任务就绪表,也就是将相应位置设置为1,同时任务就绪组也做相应的改变。

任务的优先级如35,写成二进制后,因为最高优先级是63,所以最高两位一定是0。去掉这两位后,三位三位地划分。高三位值为4,查OSMapTbl[4],得到00010000,1的位置刚好是OSRdyGrp中该置位的位置。为了不影响其他位,用位或操作符(|)来置相应位。高三位通过右移运算符(>>)得到。

低三位通过与0x07进行位与运算得到值为3。查OSMapTbl[3],得到00001000,1的位置刚好是OSRdyTbl[4]中该置位的位置。同样,为了不影响其他位,也用位或操作符(|)置相应位。

因此,任务产生或者任务进入就绪可以用以下语句实现:

```
OSRdyGrp| = OSMapTbl[priority >> 3];
OSRdyTbl[priority >> 3]| = OSMapTbl[priority&0x07];
```

(2) 任务删除/退出就绪。

任务删除或者退出就绪都需要把相应优先级的就绪表和就绪组清零。具体代码如下:

```
If((OSRdyTbl[priority >> 3]& = ~OSMapTbl[priority&0x07]) == 0)
    OSRdyGrp & =  ~OSMapTbl[priority >> 3];
```

与进入就绪状态略有不同的是,退出时就绪组相应位是否要改变,需要看就绪表中该行是否还有非零元素。因此对应代码的是一个if结构语句。

```
if((OSRdyTbl[priority >> 3] & =  ~OSMapTbl[priority & 0x07]) =  = 0)
```

这个语句表示先运算OSRdyTbl[priority >> 3] &=~OSMapTbl[priority & 0x07],再进行逻辑判断,判断OSRdyTbl[priority >> 3]是否等于0。参考前面的讲述,运算时利用与操作使得就绪表中相应位被清零。判断后,如果该行全部为0,才将就绪组中相应位清零。

(3) 获得任务最高优先级。

如前所述,将就绪组的值拿来做决策表的下标,可以查到目前最高优先级任务的优先级高三位。用高三位找到就绪表对应的那一行,将该行的值做下标查决策表就可以得到最高优先级的低三位,将高低位组合在一起就获得了当前就绪任务中最高优先级的任务优先级。依靠这个优先级,可以通过优先级数组找到该优先级对应的任务控制块,从而可以进行操作,把MCU的控制权交给这个任务去运行。操作代码如下:

```
High3Bit =  OS UnMapTbl[OSRdyGrp];
```

```
Low3Bit  = OSUnMapTbl[OSRdyTbl[high3Bit]];
priority = (hig3Bit << 3) + low3Bit;
```

5.2.2 μCOS-Ⅱ系统初始化

μCOS-Ⅱ的初始化是利用 OSInit()函数实现的。在使用 μCOS 的所有服务之前，必须调用 OSInit()，对 μCOS 自身的运行环境进行初始化。

OSInit()运行如下函数：

1. OS_InitMisc()

初始化一些全局变量。

```
OSTime              = 0L          系统当前时间(节拍数)
OSIntNesting        = 0           中断嵌套的层数
OSLockNesting       = 0           调用 OSSchededLock 的嵌套层数
OSTaskCtr           = 0           已建立的任务数
OSRunning           = FALSE       判断系统是否正在运行的标志
OSCtxSwCtr          = 0           上下文切换的次数
OSIdleCtr           = 0L          空闲任务计数器
OSIdleCtrRun        = 0L          空闲任务每秒的计数值
OSIdleCtrMax        = 0L          空闲任务每秒计数的最大值
OSStatRdy           = FALSE       统计任务是否就绪
```

2. OS_InitRdyList()

将就绪表及相关变量清零。

3. OS_InitTCBList()

建立任务控制块 TCB 链表，OSTCBList 用于指向这个链表，链表中的每个节点存放每个任务的信息(优先级、堆栈指针等)。

OSTCBPrioTbl[]是一个指针数组，指向每个任务节点，方便快速定位。

4. OS_InitEventList()

建立事件控制块 ECB 链表，链表中的每个节点存放每个事件的类型(信号量、互斥量等)、计数、等待任务表等。

ECB 空闲链表如图 5.8 所示。系统初始化后，在系统中建立起一个空闲事件控制块链表。当用户程序中请求建立新任务时，就可以从这个链表上直接摘取一个空闲控制块填写相应的信息。系统中有最大任务数限制，所以这个链表长等于最大任务数。

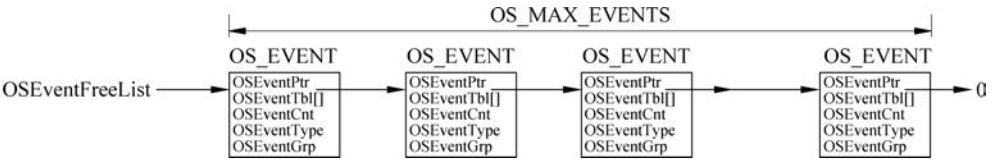

图 5.8 ECB 空闲链表

5. OS_FlagInit

事件标志初始化。事件标志是用来做多个任务逻辑并发触发一个新任务的。该数据结构是 μCOS-Ⅱ中设计的一种特有结构。通过事件标志，可以用多个任务“并”或“或”的方式来引起另一个任务的触发。

6. OS_MemInit

内存初始化后，所有的可分空闲内存用内存块空闲链表(见图 5.9)的方式连起来。当有内存使用申请时，则从链表上摘取内存块填写相应信息，分配给任务。

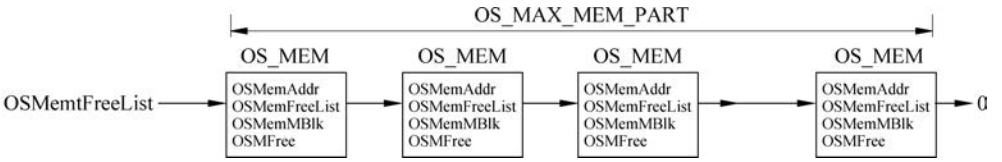

图 5.9　内存块空闲链表

7. OS_QInit

邮箱初始化。邮箱是任务间进行通信的一种方式，因此也有一个自己的数据结构。邮箱空闲队列链表如图 5.10 所示。

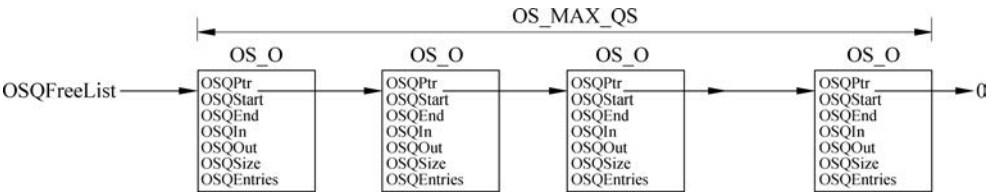

图 5.10　邮箱空闲队列链表

8. OS_InitTaskIdle

建立空闲任务。该任务必须建立，即系统必须至少有一个任务运行，该任务只做简单的计数工作。

9. OS_InitTaskStat

建立统计任务，可选 OS_TASK_STAT_EN＞0，用于计算当前 MCU 利用率。MCU 的利用率(%)＝100×(1－OSIdleCtr/OSIdleCtrMax)。

5.2.3　μCLinux 的系统初始化

μCLinux 自带的引导程序加载内核。该引导程序代码在 Linux/arch/armnommu/boot/compressed 目录下。其中，head.s 的作用最关键，它完成了加载内核的大部分工作；misc.c 则提供加载内核所需要的子程序，其中解压内核的子程序是 head.s 调用的重要程序；另外，加载内核还必须知道系统必要的硬件信息，该硬件信息在 hardware.h 中并被 head.s 所引用。

当 BootLoader 将控制权交给内核的引导程序时，第一个执行的程序就是 head.s。下面介绍 head.s 加载内核的主要过程：head.s 首先切换模式，屏蔽中断，再配置系统寄存器，再初始化 ROM、RAM 以及总线等控制寄存器，设置 Flash 和 SDRAM 的地址范围（如 ARM 中设为 0x000000～0x200000 和 0x1000000～0x2000000）；接着将内核映像（image）文件从 Flash 复制到 SDRAM，并将 Flash 和 SDRAM 的地址区间分别重映像；然后调用 misc.c 中的解压内核函数（decompress_kernel），对复制到 SDRAM 的内核映像文件进行解压缩；最后跳转到 start-kernel 执行调用内核函数（call_kernel），将控制权交给解压后的 μCLinux 系统。head.s 文件中程序流程如图 5.11 所示。decompress_kernel 解压缩函数流程如图 5.12 所示。

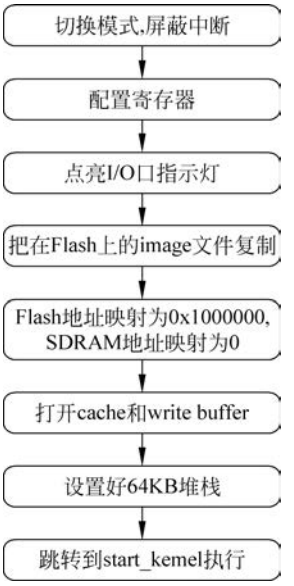

图 5.11 head.s 文件中程序流程

执行 call_kernel 函数实际上是执行 Linux/init/main.c 中的 start_kernel 函数，包括处理器结构的初始化、中断的初始化、进程相关的初始化以及内存初始化等重要工作。start_kernel()流程如图 5.13 所示。

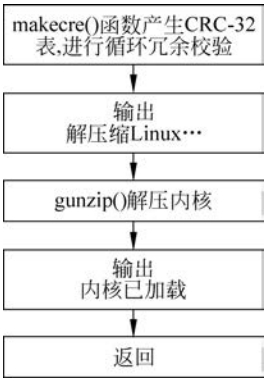

图 5.12 decompress_kernel 解压缩函数流程

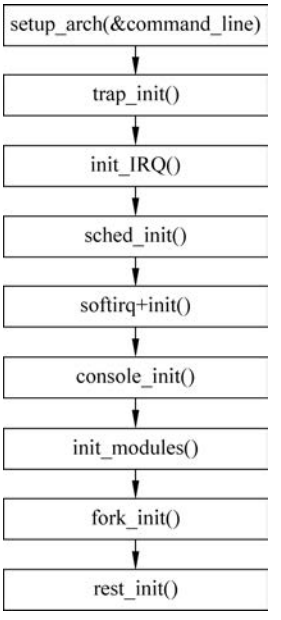

图 5.13 start_kernel()流程

main.c 中的 start_kernel 函数主要完成硬件设备初始化并为程序的执行建立环境，功能列举如下：

（1）setup_arch：体系结构初始化，根据不同的体系结构进行不同的初始化。

（2）parse_options：分析内核命令行参数，μCLinux 启动时有时需要命令行参数。这里将分析这些参数，以备将来使用。

（3）trap_init：设置内部中断，该中断由处理器使用。

（4）init_IRQ：设置外部中断，该中断是外设的中断，由用户使用。

（5）sched_init：与进程相关的初始化。

（6）time_init：时钟初始化。

（7）console_init：初始化控制台设备；控制台主要用于系统提示信息的输出。

（8）init_modules：准备内核模块。

一些内存管理的函数，包括缓存的设置等；在内存中建立各个缓冲 Hash 表，为 kernel 对文件系统的访问做准备。相关名词如下：

（1）dentry：目录数据结构。

（2）inode：i 节点。

（3）mount cache：文件系统加载缓冲。

（4）buffer cache：内存缓冲区。

（5）page cache：页缓冲区。

dentry 目录数据结构（目录入口缓存）提供了一个将路径名转化为特定的 dentry 的快的查找机制，dentry 只存在于 RAM 中。i 节点（inode）数据结构存放磁盘上的一个文件或目录的信息，i 节点存在于磁盘驱动器上。存在于 RAM 中的 i 节点就是 VFS（Virtual File System，虚拟文件系统）的 i 节点，dentry 所包含的指针指向的就是它。buffer cache 内存缓冲区用来在内存与磁盘间做缓冲处理。

准备进程需要的数据结构，然后启动第一个进程 init。这是系统中的第一个进程，其进程号（PID）永远为 1，是被用来定义系统运行级别的。init 函数流程如图 5.14 所示。启动 init 标志着用户模式（user_mode）开始。

随后可以开始初始化 PCI（Peripheral Component Interconnect，外设部件互联）和 Socket，启动交换守护进程，加载块设备驱动等。

图 5.14　init 函数流程

习题

1. 简述嵌入式操作系统的 BootLoader 的编写步骤。
2. 阅读某操作系统的 BootLoader 程序，了解 BootLoader 中使用的主要数据结构。
3. BootLoader 后系统的存储空间是怎样的？
4. 简述 BootLoader 是如何引导嵌入式操作系统开始运行的。