第 3 章

OpenHarmony启动

LiteOS-M 操作系统的启动通常包含以下阶段：

(1) Bootloader 阶段，即操作系统加载阶段，操作系统启动之前，通常运行位于固件上的操作系统加载程序 Bootloader，完成一些底层硬件的初始化工作，为启动操作系统所需的基本环境做准备，然后加载操作系统内核。

(2) 内核启动(Kernel Startup)与系统初始化(System Initialization)阶段，内核启动阶段，操作系统内核完成处理器配置、中断向量设置。系统初始化阶段，完成系统时钟的初始化。然后调用 main 完成其余初始化。

(3) main 阶段，即主程序阶段，调用 main.c 的 main()函数，调用 LOS_KernelInit()初始化内核，创建用户任务，调用 LOS_Start()开始任务调度。

(4) LOS_KernelInit 阶段，即内核初始化阶段，完成操作系统注册、内存初始化等工作。

(5) 用户程序初始化阶段，创建用户任务并初始化。

(6) 任务调度阶段，即 LOS_Start 阶段，开始任务调度。

其中 Bootloader 阶段是固件加载操作系统内核前的工作，本章不进行介绍，本章主要对启动阶段 2～6 进行分析。

3.1 内核启动与系统初始化阶段

内核启动阶段即 Kernel Startup 阶段，该阶段一般执行名为 startupXXX.S 的汇编代码，而本书的 virt 系统执行的是 liteos_m\targets\riscv_virt32 \los_start.S 汇编代码。

los_start.S 设置中断向量表等，然后调用 main()函数，如引用 3.1 所示。主要过程如下：

(1) 定义 PUSH_CALLER_REG 函数用于保存调用者负责保存的寄存器。(18～36 行)

(2) 声明了外部函数 memset。(38 行)

（3）定义了全局变量_start，该变量在 Liteos.ld 中语句 ENTRY(_start)将_start 作为内核的入口。（39 行）

（4）将常量 RISCV_MSTATUS_MPP 放入 t0 寄存器。（43 行）

（5）用 t0 寄存器设置 M 模式状态寄存器 mstatus。（44 行）

（6）M 模式中断使能寄存器 mie 清零，即关中断。（45 行）

（7）将陷入向量 HalTrapVector 放入 t0 寄存器。（46 行）

（8）用 t0 寄存器设置 M 模式陷入向量寄存器 mtvec。（47 行）

（9）保存编译选项。（49 行）

（10）设置编译选项 norelax，不支持链接器松弛，松弛链接需要 gp 寄存器，51 行设置 gp 寄存器，因此暂时关闭链接器松弛。链接器松弛在 RISC-V 平台用于优化符号地址的处理和分支跳转等。链接时如果符号地址在 gp±2KB 内，就替换掉 lui 或 puipc 指令的绝对地址或 pc 相对寻址，变为 gp 相对寻址，使得代码效率更高。该过程被称为链接器松弛(Linker Relaxation)，可以使用“-Wl,--no-relax”选项来关闭此功能。地址__global_pointer 加载到 gp(Global Pointer)全局指针寄存器，该寄存器用于优化±2KB 范围内全局变量的访问；__global_pointer 在 Liteos.ld 中定义，在数据段开始位置后 0x800 的位置。（50～51 行）

（11）恢复编译选项。（52 行）

（12）将 bss 段清零，__bss_start 和 __bss_end 在 Liteos.ld 中定义。设置堆栈指针为__start_and_irq_stack，__start_and_irq_stack_top 在 Liteos.ld 中定义，是系统启动时的堆栈。（54～59 行）

（13）调用 main 函数(main.c 中定义)。（63 行）

引用 3.1　设置中断向量表(los_start.S)

```
16  #include "soc.h"
17
18  .macro PUSH_CALLER_REG
19      addi  sp,sp,-(INT_SIZE_ON_STACK)
20      SREG  t6,16 * REGBYTES(sp)
21      SREG  t5,17 * REGBYTES(sp)
22      SREG  t4,18 * REGBYTES(sp)
23      SREG  t3,19 * REGBYTES(sp)
24      SREG  a7,20 * REGBYTES(sp)
25      SREG  a6,21 * REGBYTES(sp)
26      SREG  a5,22 * REGBYTES(sp)
27      SREG  a4,23 * REGBYTES(sp)
28      SREG  a3,24 * REGBYTES(sp)
29      SREG  a2,25 * REGBYTES(sp)
30      SREG  a1,26 * REGBYTES(sp)
31      SREG  a0,27 * REGBYTES(sp)
32      SREG  t2,28 * REGBYTES(sp)
33      SREG  t1,29 * REGBYTES(sp)
34      SREG  t0,30 * REGBYTES(sp)
35      SREG  ra,31 * REGBYTES(sp)
36  .endm
37
38  .extern memset
```

```
39  .global _start
40  .section .start.text
41  .align 4
42  _start:
43      li    t0,RISCV_MSTATUS_MPP
44      csrw  mstatus,t0
45      csrw  mie,zero
46      la    t0,HalTrapVector
47      csrw  mtvec,t0       # direct mode
48
49      .option push
50      .option norelax
51      la    gp,__global_pointer$
52      .option pop
53
54      la    t0,__bss_start
55      la    t1,__bss_end
56  2:
57      sw    zero,0x0(t0)
58      addi  t0,t0,0x4
59      bgtu  t1,t0,2b
60
61      la    sp,__start_and_irq_stack_top
62
63      tail  main
```

3.2　主程序阶段

主程序阶段，调用 main.c 的 main()函数。代码见\liteos_m\targets\riscv_virt32\main.c，如引用 3.2 所示。main()函数一般先调用 LOS_KernelInit()函数完成内核初始化，然后调用 LosAppInit()函数初始化建用户程序，例如 test_demo.c，最后调用 LOS_Start()函数开始任务调度。

引用 3.2　主程序阶段(main.c)

```
73  LITE_OS_SEC_TEXT_INIT INT32 main(VOID)
74  {
75      UINT32 ret;
76
77      printf("\n OHOS start \n\r");
78
79      ret = LOS_KernelInit();
80      if (ret != LOS_OK) {
81          printf("Liteos kernel init failed! ERROR: 0x%x\n",ret);
82          goto START_FAILED;
83      }
84
85      ret = LosAppInit();
86      if (ret != LOS_OK) {
87          printf("LosAppInit failed! ERROR: 0x%x\n",ret);
```

```
88        }
89
90        LOS_Start();
91
92    START_FAILED:
93        while (1) {
94            __asm volatile("wfi");
95        }
96    }
97
```

3.3 内核初始化阶段

内核初始化阶段即 LOS_KernelInit 阶段，该阶段根据配置宏等调用以下函数完成各子系统初始化：

(1) OsRegister()；　　登记最大任务数。
(2) OsMemSystemInit()；　　调用 LOS_MemInit()(los_memory.c)初始化内存子系统。
(3) HalArchInit()；　　调用 HalHwiInit()(los_interrupt.c)完成硬件中断初始化。
(4) OsTaskInit()；　　任务子系统初始化(los_task.c)。
(5) OsTaskMonInit()；　　任务监控子系统初始化(los_task.c)。
(6) OsCpupInit()；　　CPU 子系统初始化(los_cpup.c)。
(7) OsSemInit()；　　信号量 Semphore 初始化(los_sem.c)。
(8) OsMuxInit()；　　互斥锁 Mux 子系统初始化(los_mux.c)。
(9) OsQueueInit()；　　消息队列子系统初始化(los_queue.c)。
(10) OsSwtmrInit()；　　软时钟子系统初始化(los_swtmr.c)。
(11) OsIdleTaskCreate()；　　空闲任务初始化(los_task.c)。
(12) LOS_TraceInit()；　　跟踪子系统初始化(los_debug.c)。
(13) OsExcMsgDumpInit()；异常信息转存子系统初始化(los_exec_info.c)。

代码见\liteos_m\kernel\src\los_init.c，如引用 3.3 所示。

引用 3.3　内核初始化阶段(los_init.c)

```
111  LITE_OS_SEC_TEXT_INIT UINT32 LOS_KernelInit(VOID)
112  {
113      UINT32 ret;
114      PRINTK("entering kernel init...\n");
115
116      OsRegister();
117      ret = OsMemSystemInit();
118      if (ret != LOS_OK) {
119          PRINT_ERR("OsMemSystemInit error %d\n",ret);
120          return ret;
121      }
122
123      HalArchInit();
```

```
124
125     ret = OsTaskInit();
126     if (ret != LOS_OK) {
127         PRINT_ERR("OsTaskInit error\n");
128         return ret;
129     }
130
131 #if (LOSCFG_BASE_CORE_TSK_MONITOR == 1)
132     OsTaskMonInit();
133 #endif
134
135 #if (LOSCFG_BASE_CORE_CPUP == 1)
136     ret = OsCpupInit();
137     if (ret != LOS_OK) {
138         PRINT_ERR("OsCpupInit error\n");
139         return ret;
140     }
141 #endif
142
143 #if (LOSCFG_BASE_IPC_SEM == 1)
144     ret = OsSemInit();
145     if (ret != LOS_OK) {
146         return ret;
147     }
148 #endif
149
150 #if (LOSCFG_BASE_IPC_MUX == 1)
151     ret = OsMuxInit();
152     if (ret != LOS_OK) {
153         return ret;
154     }
155 #endif
156
157 #if (LOSCFG_BASE_IPC_QUEUE == 1)
158     ret = OsQueueInit();
159     if (ret != LOS_OK) {
160         PRINT_ERR("OsQueueInit error\n");
161         return ret;
162     }
163 #endif
164
165 #if (LOSCFG_BASE_CORE_SWTMR == 1)
166     ret = OsSwtmrInit();
167     if (ret != LOS_OK) {
168         PRINT_ERR("OsSwtmrInit error\n");
169         return ret;
170     }
171 #endif
172
173     ret = OsIdleTaskCreate();
174     if (ret != LOS_OK) {
175         return ret;
176     }
```

```
177
178  #if (LOSCFG_KERNEL_TRACE == 1)
179      ret = LOS_TraceInit();
180      if (ret != LOS_OK) {
181          PRINT_ERR("LOS_TraceInit error\n");
182          return ret;
183      }
184  #endif
185
186  #ifdef LOSCFG_TEST
187      //ret = los_TestInit();
188      //if (ret != LOS_OK) {
189      //    PRINT_ERR("los_TestInit error\n");
190      //    return ret;
191      //}
192  #endif
193
194  #if (LOSCFG_PLATFORM_EXC == 1)
195      OsExcMsgDumpInit();
196  #endif
197
198      return LOS_OK;
199  }
200
```

3.4 用户程序初始化与任务调度

1. 用户程序初始化

main 函数调用 LosAppInit()初始化用户程序，如引用 3.4 所示。主要完成用户任务的创建。

引用 3.4 用户程序初始化阶段（test_demo.c）

```
32  #include "los_task.h"
33  #include "los_debug.h"
34
35  static void TaskSampleEntry2(void)
36  {
37      while(1) {
38          //printf("TaskSampleEntry2 running...\n\r");
39          printf("Task2\n\r");
40          LOS_TaskDelay(1000);
41      }
42  }
43
44
45  static void TaskSampleEntry1(void)
46  {
47      while(1) {
48          //printf("TaskSampleEntry1 running...\n\r");
```

```
49            printf("Task1\n\r");
50            LOS_TaskDelay(1000);
51        }
52    }
53
54    unsigned int LosAppInit(VOID)
55    {
56        unsigned int ret;
57        unsigned int taskID1,taskID2;
58        TSK_INIT_PARAM_S task1 = { 0 };
59        task1.pfnTaskEntry = (TSK_ENTRY_FUNC)TaskSampleEntry1;
60        task1.uwStackSize  = 0x1000;
61        task1.pcName       = "TaskSampleEntry1";
62        task1.usTaskPrio   = 6;
63        ret = LOS_TaskCreate(&taskID1,&task1);
64        if (ret != LOS_OK) {
65            printf("Create Task failed! ERROR: 0x%x\n",ret);
66            return ret;
67        }
68
69        task1.pfnTaskEntry = (TSK_ENTRY_FUNC)TaskSampleEntry2;
70        task1.uwStackSize  = 0x1000;
71        task1.pcName       = "TaskSampleEntry2";
72        task1.usTaskPrio   = 7;
73        ret = LOS_TaskCreate(&taskID2,&task1);
74        if (ret != LOS_OK) {
75            printf("Create Task failed! ERROR: 0x%x\n",ret);
76        }
77
78        return ret;
79    }
80
```

2. 任务调度

main()函数最后调用 LOS_Start()函数开始任务调度，如引用 3.5 所示。

引用 3.5　开始任务调度(los_init.c)

```
99    LITE_OS_SEC_TEXT_INIT UINT32 LOS_Start(VOID)
100   {
101       return HalStartSchedule(OsTickHandler);
102   }
103
```

3.5　编程实践三：启动分析

首先 VS Code 中运行任务 LiteOS_M：QemuForDebugger 开启 QEMU 供调试。在新的终端下，打开 gdb 进行调试，调试命令如下：

```
cd /mnt/d/liteos_m/targets/riscv_virt32/GCC
make debug
```

如果显示“(gdb)”则已经进入 gdb 中，可以进行调试。

3.5.1 跟踪调试 los_start.S

1. 第一条指令的位置

内核的汇编代码 liteos.dasm 文件部分代码如引用 3.6 所示，_text_start 是 kernel 的第一条指令，按照文件 los_start.S 的对应，_text_start 也即全局变量_text 的位置，按照 Liteos.ld，它也是内核的入口地址。

引用 3.6 内核的部分汇编代码（liteos.dasm）

```
1
2   build/liteos.elf:     file format elf32-littleriscv
3
4
5   Disassembly of section .text:
6
7   80000000 <__text_start>:
8   80000000:000022b7              lui    t0,0x2
9   80000004:80028293 addi   t0,t0,-2048 # 1800 <EXCEPT_STACK_SIZE+0x1000>
10  80000008:30029073 csrw   mstatus,t0
11  8000000c:30401073 csrw   mie,zero
12  80000010:297 auipc t0,0x0
13  80000014:19028293 addi   t0,t0,400 # 800001a0 <HalTrapVector>
14  80000018:30529073 csrw   mtvec,t0
15  8000001c:7197 auipc gp,0x7
16  80000020:aa418193              addi gp,gp,-1372 # 80006ac0 <__global_pointer$>
17  80000024:b2018293              addi t0,gp,-1248 # 800065e0 <__data_end>
18  80000028:ec018313              addi t1,gp,-320 # 80006980 <__bss_end>
19  8000002c:0002a023              sw    zero,0(t0)
20  80000030:291  addi t0,t0,4
21  80000032:fe62ede3              bltu t0,t1,8000002c <__text_start+0x2c>
22  80000036:9117 auipc sp,0x9
23  8000003a:14a10113              addi sp,sp,330 # 80009180 <__start_and_irq_stack_top>
24  8000003e:a44d                    j    800002e0 <main>
25
```

思考题 1：给出文件 liteos_start.S 的对全局变量_text 的定义。

思考题 2：给出文件 Liteos.ld，它也是内核的入口地址。

2. los_start.S 的跟踪

在_start 处设置断点。

```
(gdb) b _start
Breakpoint 1 at 0x80000008
(gdb) s
Cannot find bounds of current function
(gdb) c
Continuing.

Breakpoint 1,0x80000008 in _start ()
```

程序运行到地址为 0x80000008 的函数_start()处停止，该函数在汇编程序 los_start.S 中定义。请读者跟踪调试_start()函数，并完成思考题 3 以及思考题 4。

思考题 3：反汇编，分析 los_start.S 中的："la　t0，__bss_start"和"la　t1，__bss_end"指令实际上是如何实现的？为什么？

思考题 4：查看陷入向量 HalTrapVector，反汇编给出其代码。

3.5.2　跟踪调试 main()函数

请读者跟踪调试 main()函数，并完成思考题 5 及思考题 6。

思考题 5：main()函数的地址是什么？在该处设置断点，列出其源程序。

思考题 6：在引用 3.2 函数 main()的 79 行设置断点，并继续运行代码到断点处，单步运行跟踪进入 LOS_KernelInit()函数。反汇编该函数，分析其进入及返回时的参数传递与堆栈处理。