

使用QEMU分析RT-Thread系统

张 戈

南京世唯通信技术有限公司, 江苏 南京

收稿日期: 2026年4月3日; 录用日期: 2026年4月13日; 发布日期: 2026年9月29日

摘 要

本文基于QEMU vexpress-a9平台, 搭建无需硬件的RT-Thread调试环境。通过arm-none-eabi-gdb远程连接, 跟踪系统上电、MMU/页表建立、堆初始化、多任务调度、中断异常与设备驱动全过程; 并对RT-Thread的初始化全流程进行分析说明。

关键词

RT-Thread, QEMU, GDB调试, 开源实时操作系统

Using QEMU to Analyze the RT-Thread System

Ge Zhang

Nanjing Shiwei Communication Co., Ltd., Nanjing Jiangsu

Received: April 3, 2026; accepted: April 13, 2026; published: September 29, 2026

Abstract

This article is based on the QEMU vexpress-a9 platform to build a hardware-free RT-Thread debugging environment. Through the remote connection of arm-none-eabi-gdb, the entire process from system power-on, MMU/page table establishment, heap initialization, multi-task scheduling, interrupt exceptions and device drivers is tracked; and the entire initialization process of RT-Thread is analyzed and explained.

Keywords

RT-Thread, QEMU, GDB Debugging, Open-Source Real-Time Operating System



1. 引言

在嵌入式系统开发中，调试环节往往受限于硬件平台、仿真器与复杂环境，难以快速定位启动、内存管理、多任务调度及驱动问题。QEMU [1] 作为开源指令级模拟器，能够完整模拟 ARM Cortex-A9 多核 SoC 行为，为 RT-Thread 提供零成本、可重复的调试与验证环境。本文基于 QEMU vexpress-a9 [2] 机器，构建一套从编译、启动到中断与设备驱动的可视化调试流程，结合 GDB 远程协议实现源码级单步、断点、寄存器与线程状态跟踪，帮助开发者在脱离真实硬件的条件下，深入理解 RT-Thread 运行机理，并可将调试结果无缝迁移到物理板卡，大幅缩短研发周期。

2. 搭建 QEMU 平台下 RT-Thread 调试环境

2.1. QEMU 简介

QEMU 是纯开源、跨平台的「系统级 + 用户级」模拟器与虚拟化引擎，可在宿主机上运行为不同 CPU 架构编译的二进制代码。

- 指令级翻译：基于 TCG (Tiny Code Generator) 动态二进制翻译，把目标指令即时转译为宿主机指令，支持 ARM/AArch64、RISC-V、MIPS、PowerPC、x86 等 20 余种架构。
- 设备模型丰富：内置串口、网卡、显示、USB、PCI-e、Flash、SD、CAN 等大量 SoC 外设，可整机模拟 Beagle/VExpress/Raspberry Pi/i.MX6 等开发板。
- 多核 SMP：支持多 CPU 核、NUMA、GIC/IO-APIC 中断控制器，满足 RT-Thread/Linux SMP 调试需求。
- 调试钩子：提供 GDB Remote、QMP、Tracing、Semihosting 接口，可断点、单步、查看寄存器/内存、注入中断，实现「零 JTAG」源码级调试。

因此 QEMU 成为嵌入式开发、系统软件验证、持续集成及教学场景中不可或缺的「通用仿真平台」。

2.2. RT-Thread 简介

RT-Thread 是一款国人自研的开源、轻量级、组件化的实时操作系统(RTOS)，诞生于 2006 年，专为嵌入式场景设计，支持资源受限的微控制器(MCU)到多核应用处理器(MPU)。其核心仅约 3 kB ROM、1.2 kB RAM，可通过模块化组件无限扩展，提供文件系统、网络协议栈、图形界面、设备框架等丰富功能，并支持 SMP、动态模块、POSIX、C++、MicroPython 等高级特性。在开源社区中我们可以直接拉取代码进行学习研究。另外，RT-Thread 可以使用 qemu 进行仿真调试，如针对 armv7 架构的 vexpress-a9 平台，以及 armv8 架构，riscv64 等都可以直接进行仿真。本文主要以 vexpress-a9 处理平台进行描述，虚拟机使用 VMware® Workstation 17 Pro 运行 Ubuntu20.04。

2.3. 仿真环境搭建

- 安装 qemu
`sudo apt install -y qemu-system-arm`
- 从 git 上拉取源码到 Ubuntu 环境(图 1):
`git clone https://gitee.com/rththread/rt-thread.git`

```

lubancat@ubuntu:~$ git clone https://gitee.com/rththread/rt-thread.git
Cloning into 'rt-thread'...
remote: Enumerating objects: 303021, done.
remote: Counting objects: 100% (5029/5029), done.
remote: Compressing objects: 100% (2242/2242), done.
remote: Total 303021 (delta 3373), reused 3550 (delta 2748), pack-reused 297992 (from 1)
Receiving objects: 100% (303021/303021), 882.73 MiB | 3.08 MiB/s, done.
Resolving deltas: 100% (222356/222356), done.
Updating files: 100% (36348/36348), done.
lubancat@ubuntu:~$

```

Figure 1. Source code pull request

图 1. 源码拉取

- 在 Ubuntu 上安装交叉编译工具 arm-none-eabi-gcc
- 更新源: `sudo apt update`;
 - 安装官方仓库版本: `sudo apt install gcc-arm-none-eabi binutils-arm-none-eabi libnewlib-arm-none-eabi` (图 2)。

```

lubancat@ubuntu:~$ arm-none-eabi-gcc --version
arm-none-eabi-gcc (15:9-2019-q4-0ubuntu1) 9.2.1 20191025 (release) [ARM/arm-9-branch revision 277599]
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

```

Figure 2. Toolchain version information

图 2. 工具链版本信息

- 配置 RT-Thread 系统

因为我们使用 vexpress-a9 平台进行仿真,需要进入 `rt-thread/bsp/qemu-vexpress-a9` 目录。因为 vexpress-a9 是双核处理器,但是内核默认配置没有打开 SMP,需要对内核进行配置(图 3)。

```

(Top) → RT-Thread Kernel
klibc options --->
(16) The maximal size of kernel object name
[ ] Use the data types defined in ARCH_CPU
[ ] Enable RT-Thread Nano
[ ] Enable RT-Thread Smart (microkernel on kernel/userland)
[ ] Enable AMP (Asymmetric Multi-Processing)
[*] Enable SMP (Symmetric multiprocessing)
(1) Number of CPUs
(8) Alignment size for CPU architecture data access
The maximal level value of priority of thread (256) --->
(100) Tick frequency, Hz
[*] Using stack overflow checking
[*] Enable system hook
[*] Using function pointers as system hook
[ ] Enable hook list
-*- Enable IDLE Task hook
(4) The max size of idle hook list
(4096) The stack size of idle thread

```

Figure 3. Kconfig configuration interface

图 3. Kconfig 配置界面

勾选上保存后,使用 `scons` 命令开始编译内核(图 4)。

```

CC build/kernel/libcpu/arm/cortex-a/pmu.o
CC build/kernel/libcpu/arm/cortex-a/stack.o
AS build/kernel/libcpu/arm/cortex-a/start_gcc.o
CC build/kernel/libcpu/arm/cortex-a/trap.o
AS build/kernel/libcpu/arm/cortex-a/vector_gcc.o
CC build/kernel/src/clock.o
CC build/kernel/src/components.o
CC build/kernel/src/cpu_mp.o
CC build/kernel/src/defunct.o
CC build/kernel/src/idle.o
CC build/kernel/src/ipc.o
CC build/kernel/src/irq.o

```

```
CC build/kernel/src/klibc/kerrno.o
CC build/kernel/src/klibc/kstdio.o
CC build/kernel/src/klibc/kstring.o
CC build/kernel/src/klibc/rt_vsnprintf_tiny.o
CC build/kernel/src/klibc/rt_vsscanf.o
CC build/kernel/src/kservice.o
CC build/kernel/src/mem.o
CC build/kernel/src/memheap.o
CC build/kernel/src/mempool.o
CC build/kernel/src/object.o
CC build/kernel/src/scheduler_comm.o
CC build/kernel/src/scheduler_mp.o
CC build/kernel/src/thread.o
CC build/kernel/src/timer.o
LINK rtthread.elf
Memory region      Used Size  Region Size  %age Used
arm-none-eabi-objcopy -O binary rtthread.elf rtthread.bin
text    data    bss    dec      hex filename
540579  35004   84128  659711  a10ff rtthread.elf
scons: done building targets.
lubancat@ubuntu:~/rt-thread/bsp/qemu-vexpress-a9$
```

Figure 4. Compilation results of source code

图 4. 源码编译结果

编译完成后就可以使用 qemu 和 gdb 工具进行单步源码级和指令级调试了(图 5)。

```
lubancat@ubuntu:~/rt-thread/bsp/qemu-vexpress-a9$ qemu-system-arm -M vexpress-a9 -snp cpus=2 -kernel rtthread.bin -nographic -sd sd.bin -s -S
WARNING: Image format was not specified for 'sd.bin' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
lubancat@ubuntu:~/rt-thread/bsp/qemu-vexpress-a9$ gdb-multiarch rtthread.elf
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from rtthread.elf...
(gdb) target remote :1234
Remote debugging using :1234
0x60000000 in ?? ()
(gdb) b main
Breakpoint 1 at 0x600101b8: file applications/main.c, line 17.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, main () at applications/main.c:17
17      printf("Hello RT-Thread!\n");
(gdb)
```

Figure 5. QEMU single-step debugging

图 5. QEMU 单步调试

3. RT-Thread 内核启动分析

3.1. RT-Thread 第一阶段启动流程分析

本小节主要介绍 RT-Thread 最开始的启动引导代码，这部分代码主要是用汇编直接写的。主要进行最初的 cpu 工作模式的建立，c 运行环境的初始化，并对 mmu 进行初始化，建立虚拟和物理内存的映射关系，这些初始化为后续 OS 各组件的初始化创建了运行前提条件。

3.1.1. 启动程序入口点

系统复位后的执行位置位于 0x60010000，此处存放中断向量表信息，第一条是复位的跳转指令(图 6)。

```
(gdb) disassemble 0x60010000
Dump of assembler code for function system_vectors:
0x60010000 <+0>:    ldr     pc, [pc, #24] ; 0x60010020 <_vector_reset>
0x60010004 <+4>:    ldr     pc, [pc, #24] ; 0x60010024 <_vector_undef>
0x60010008 <+8>:    ldr     pc, [pc, #24] ; 0x60010028 <_vector_swi>
0x6001000c <+12>:   ldr     pc, [pc, #24] ; 0x6001002c <_vector_pabt>
0x60010010 <+16>:   ldr     pc, [pc, #24] ; 0x60010030 <_vector_dabt>
0x60010014 <+20>:   ldr     pc, [pc, #24] ; 0x60010034 <_vector_resv>
0x60010018 <+24>:   ldr     pc, [pc, #24] ; 0x60010038 <_vector_irq>
0x6001001c <+28>:   ldr     pc, [pc, #24] ; 0x6001003c <_vector_fiq>
End of assembler dump.
(gdb)
```

Figure 6. Vector table information

图 6. 向量表信息

_vector_reset 处放置 32 位数值_reset 的绝对地址，此处的跳转直接让 PC 跳转到 _reset 位置。

3.1.2. 主核初始化 BSS 初始化，建立当前模式使用的栈

主核获取 cpu id, id 如果是 0，则代表当前是主 cpu，进入 normal_setup (图 7)

```
89          /* disable the data alignment check */
90          mrc p15, 0, r1, c1, c0, 0
91          bic r1, #(1<<0)          /* Disable MMU */
92          bic r1, #(1<<1)          /* Disable Alignment fault checking */
93          bic r1, #(1<<2)          /* Disable data cache */
94          bic r1, #(1<<11)         /* Disable program flow prediction */
95          bic r1, #(1<<12)         /* Disable instruction cache */
96          bic r1, #(3<<19)         /* bit[20:19] must be zero */
97          mcr p15, 0, r1, c1, c0, 0
98
99          @ get cpu id, and subtract the offset from the stacks base address
100         bl rt hw_cpu_id
>101        mov r5, r0
102
103         cmp     r5, #0             @ cpu id == 0
104         beq     normal_setup
105
106         @ cpu id > 0, stop or wait
107         #ifdef RT_SMP_AUTO_BOOT
108         ldr r0, =secondary_cpu_entry
109         mov r1, #0
110         str r1, [r0] /* clean secondary_cpu_entry */
111         #endif /* RT_SMP_AUTO_BOOT */
112
113         secondary_loop:
114         @ cpu core 1 goes into sleep until core 0 wakeup it
115         wfe
```

Figure 7. Source code of reset jump

图 7. 复位跳转源码

- stack_setup: 建立多种模式下的栈，Armv7-A 架构中包含多种模式[3]，不同模式使用不同的 sp 寄存器，这里对 SVC, Undefined Instruction Mode, Abort Mode, FIQ Mode, IRQ Mode 下的 sp 进行了初始化。SVC 模式下的 sp 位于 stack_top 位置。
- bss_loop: 对 BSS 端空间内容进行清零，BSS 端开始和结束地址位于 __bss_start 和 __bss_end，该符号位置在链接脚本中定义(图 8)。

3.1.3. 页表初始化[4]

Vexpress 平台带有 MMU 单元，RT-Thread 在运行过程中，对地址访问采用虚拟地址，因此需要事先对使用的物理地址建立虚拟映射。这里只使用到了一级页表，每个页表项映射的物理页帧大小为 1 MB。

页目录的位置位于 MMU Table。页表是实现虚拟地址到物理地址转换的查询表，在 MMU 使能后，PC 寄存器访问的地址都是虚拟地址，在页表的配合参与下，转换成物理地址，才能访问到内存上的程序指令。页表中除了地址转换的信息外，还有一些权限属性，缓存性质等，用来标记地址访问的行为。armv7-A 架构中的一级页表[5]，如下图 9 所示。

```
124      normal_setup:
125      /* setup stack */
126      bl      stack_setup
127
128      /* clear .bss */
129      mov     r0,#0
130      ldr     r1,=__bss_start
131      ldr     r2,=__bss_end
132
133      bss_loop:
134      cmp     r1,r2
135      strlo   r0,[r1],#4
136      blo     bss_loop
137
138      #ifdef RT_USING_SMP
139      mrc     p15, 0, r1, c1, c0, 1
140      mov     r0, #(1<<6)
141      orr     r1, r0
142      mcr     p15, 0, r1, c1, c0, 1 //enable smp
143      #endif
144
145      /* enable branch prediction */
146      mrc     p15, 0, r0, c1, c0, 0
147      orr     r0, r0, #(1<<11)
148      mcr     p15, 0, r0, c1, c0, 0
149
150      /* initialize the mmu table and enable mmu */
151      ldr     r0, =platform_mem_desc
152      ldr     r1, =platform_mem_desc_size
```

Figure 8. Stack initialization code, BSS initialization code
图 8. 栈初始化 BSS 初始化代码

	31	24		23	20			19	18	17	16	15	14	12		11	10	9	8	5		4	3	2	1	0	
Fault	IGNORE																									0	0
Page table	Page table base address, bits [31:10]														I M P	Domain	S B Z	N S	S B Z	0	1						
Section	Section base address, PA[31:20]				N S	0	n G	S	A P [2]	TEX [2:0]	AP [1:0]	I M P	Domain	X N	C	B	1	0									
Supersection	Supersection base address PA[31:24]		Extended base address PA[35:32]		N S	1	n G	S	A P [2]	TEX [2:0]	AP [1:0]	I M P	Extended base address PA[39:36]		X N	C	B	1	0								
Reserved	Reserved																									1	1

Figure 9. ARMv7-A level 1 page table type
图 9. ARMv7-A 一级页表类型

- 一级页表有以上 4 种类型，这里建立的一级页表项是第 3 种。PA[31:20]为物理页框的基地址(图 10、图 11)。
- `rt_hw_init_mmu_table`: 该函数是用 c 语言实现，因为在前面我们已经设置了 `svc` 模式下的 `sp` 寄存器，实现了对 c 语言运行环境初始化，这里就可以直接使用 c 语言编写代码了，该函数根据 `platform_mem_desc` 数组中描述的映射区间，建立起映射关系。
 - `platform_mem_desc`: 该数组对映射信息进行描述，包含两个区域。

```

144
145      /* enable branch prediction */
146      mrc p15, 0, r0, c1, c0, 0
147      orr    r0, r0, #(1<<11)
148      mcr p15, 0, r0, c1, c0, 0
149
150      /* initialize the mmu table and enable mmu */
151      ldr r0, =platform_mem_desc
152      ldr r1, =platform_mem_desc_size
153      ldr r1, [r1]
154      bl rt_hw_init_mmu_table
155      bl rt_hw_mmu_init
156
157      /* start RT-Thread Kernel */
158      ldr    pc, _rtthread_startup
159      _rtthread_startup:
160      .word rtthread_startup
161
162      stack_setup:
163      ldr    r0, =stack_top
164

```

Figure 10. Assembly code for page table initialization

图 10. 页表初始化汇编代码

```

struct mem_desc platform_mem_desc[] = {
    {0x10000000, 0x50000000, 0x10000000, DEVICE_MEM},
    {0x60000000, 0xe0000000, 0x60000000, NORMAL_MEM}
};

```

Figure 11. Range information of physical page configuration properties

图 11. 物理页面配置属性范围信息

物理地址 0x10000000-0x50000000 映射到虚拟地址 0x10000000-0x50000000 结束, 地址属性为 device 属性;

物理地址 0x60000000-0xe0000000 映射到虚拟地址 0x60000000-0xe0000000 结束, 地址属性为 mem 属性;

- `rt_hw_mmu_init`: 这里开启 mmu, 并打开指令和数据 cache, 同时, 将 MMU Table 地址设置到 `ttbr0` 寄存器中, `ttbr0` 是页目录基址寄存器。这里的地址访问只使用 `ttbr0`, 没有使用到 `ttbr1` (图 12)。

```

163
164      void rt_hw_mmu_init(void)
165      {
166          rt_cpu_dcache_clean_flush();
167          rt_cpu_icache_flush();
168          rt_hw_cpu_dcache_disable();
169          rt_hw_cpu_icache_disable();
170          rt_cpu_mmu_disable();
171
172          /*rt_hw_cpu_dump_page_table(MMUTable);*/
173          rt_hw_set_domain_register(0x55555555);
174
175          rt_cpu_tlb_set(MMUTable);
176
177          rt_cpu_mmu_enable();
178
179          rt_hw_cpu_icache_enable();
180          rt_hw_cpu_dcache_enable();
181      }
182

```

Figure 12. MMU initialization code

图 12. MMU 初始化代码

3.2. RT-Thread 第二阶段启动流程分析

此阶段从 `rtthread_startup` 函数开始, 该函数在 RT-Thread 中的作用相当于 Linux 内核中的 `start_kernel` 函数, 主要进行板级 BSP 初始化, RT-Thread 核心组件的初始化, 并最终把创建的线程运行起来(图 13)。

```

233     int rtthread_startup(void)
234     {
235         rt_hw_interrupt_disable();
236
237         /* board level initialization
238          * NOTE: please initialize heap inside board initialization.
239          */
>240         rt_hw_board_init();
241
242         /* show RT-Thread version */
243         rt_show_version();
244
245         /* timer system initialization */
246         rt_system_timer_init();
247
248         /* scheduler system initialization */
249         rt_system_scheduler_init();
250
251         #ifdef RT_USING_SIGNALS
252             /* signal system initialization */
253             rt_system_signal_init();
254         #endif /* RT_USING_SIGNALS */
255
256         /* create init_thread */
257         rt_application_init();
258

```

Figure 13. Initialization entry function of the second phase

图 13. 第二阶段初始化入口函数

```

40     */
41     void rt_hw_board_init(void)
>42     {
43         /* initialize hardware interrupt */
44         rt_hw_interrupt_init();
45         /* initialize system heap */
46         rt_system_heap_init(HEAP_BEGIN, HEAP_END);
47
48         rt_components_board_init();
49         rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
50
51         rt_thread_idle_sethook(idle_wfi);
52
53         #ifdef RT_USING_SMP
54             /* install IPI handle */
55             rt_hw_ipi_handler_install(RT_SCHEDULE_IPI, rt_scheduler_ipi_handler);
56         #endif
57     }
58

```

Figure 14. Board-level initialization of the vexpress Platform

图 14. vexpress 平台板级初始化

3.2.1. RT-Thread 板级初始化阶段

板级初始化与具体 CPU 平台有关(图 14), 板级主要对 cpu 内部使用的具体硬件进行初始化, 如中断控制器, 系统堆区域, 串口打印, 系统时钟 tick 等。

- `rt_hw_interrupt_init()`:
 - ① 把异常向量表设置到 VBAR;

- ② 初始化中断控制器(GICv2 的 distributor/cpu 接口);
- ③ 初始化 `isr_table` 数组, 该数组管理注册的中断信息;
- `rt_system_heap_init(HEAP_BEGIN, HEAP_END)`:
`HEAP_BEGIN` 定义为 `__bss_end`, 该符合由链接脚本给出, 位于 `bss` 段的结束位置。`HEAP_END` 可以根据实际内存大小自定义。这里划定 `heap` 的区间, 建立 RT-Thread 的堆管理器(`rt_malloc/rt_free` 基石)。
- `rt_components_board_init()`:
 把 `INIT_EXPORT()`宏声明的初始化函数按链接顺序全部跑一遍, 此时调度器尚未启动, 所以只能做“不能阻塞、不能依赖线程”的纯硬件/驱动初始化(图 15)。

```
#define INIT_EXPORT(fn, level)
RT_USED const init_fn_t __rt_init_##fn SECTION(".rti_fn." level) = fn
```

Figure 15. INIT_EXPORT macro definition

图 15. INIT_EXPORT 宏定义

`INIT_EXPORT()`宏修饰的函数, 通过定义函数指针, 并把指针链接到一个特定区域中, 初始化程序找到该区域依次调用函数指针, 完成函数的初始化。

- `rt_console_set_device(RT_CONSOLE_DEVICE_NAME)`:
`RT_CONSOLE_DEVICE_NAME` 默认是“uart0”(可在 `rtconfig.h` 改)。
 把 `rt_kprintf()`的底层输出绑定到指定串口设备, 此后 `rt_kprintf()`真正可用。
- `rt_thread_idle_sethook(idle_wfi)`:
`idle_wfi` 实现是 `asm volatile("wfi")`; 让空闲线程每次循环都执行 `wfi(wait-for-interrupt)`, 降低功耗。
- `rt_hw_ipi_handler_install(RT_SCHEDULE_IPI, rt_scheduler_ipi_handler)`:
 向 GIC 注册核间中断号 `RT_SCHEDULE_IPI`(通常为 0 或 1)的回调。当 CPU0 想让 CPU1 做调度时, 使用 `rt_hw_ipi_send(cpu_mask, RT_SCHEDULE_IPI)`向 CPU1 发送一个软件 IPI, CPU1 在中断里执行 `rt_scheduler_ipi_handler()`完成跨核任务抢占/迁移。

3.2.2. RT-Thread 组件初始化

板级初始化完成后, RT-Thread 就可以进行内核核心组件的始化了, 核心组件是更抽象的一层概念, 基于 CPU 板级硬件提供的功能, 完成操作系统组件的初始化。

- `rt_system_timer_init()`: 初始化系统定时器。系统定时器在回调函数在 `tick` 中断产生后进行检测, 系统定时器的回调函数是在中断上下文执行, 因此回调函数不能使用引起阻塞的 API。
- `rt_application_init()`: 创建 `main` 线程, `main` 线程运行时, 调用 `rt_components_init()`对系统组件进行初始化, 进而调用 `rt_hw_secondary_cpu_up` 函数引导 CPU1 启动, 最后执行自定义的 `main` 函数, 在 `main` 函数体中, 用户可以自定义自己的业务逻辑(图 16)。
- `rt_system_timer_thread_init()`: 初始化软件定时器, 并创建软件定时器线程。定时器线程在回调函数检测到软件定时器到期后, 执行定时器的回调函数。由于是在进程上下文执行, 回调函数可以阻塞。
- `rt_thread_idle_init()`: 创建 `tidle` 线程, 每个 CPU 核心有独立的 `tidle` 线程。由于 `tidle` 线程优先级比较低, 一般在系统中没有其他任务时, 才会执行 `tidle` 线程。
- `rt_system_scheduler_start()`:
 - ① 从就绪队列中选出优先级最高的任务
 - ② 把 CPU 控制权交给该任务, 从此开始真正的多线程调度(图 17)。

```

    */
    void main_thread_entry(void *parameter)
    {
        extern int main(void);

#ifdef RT_USING_COMPONENTS_INIT
        /* RT-Thread components initialization */
        rt_components_init();
#endif /* RT_USING_COMPONENTS_INIT */

#ifdef RT_USING_SMP
        rt_hw_secondary_cpu_up();
#endif /* RT_USING_SMP */
        /* invoke system main function */
#ifdef __ARMCC_VERSION
        {
            extern int $$Super$$main(void);
            $$Super$$main(); /* for ARMCC. */
        }
#elif defined(__ICCARM__) || defined(__GNUC__) || defined(__TASKING__)
        main();
#endif
    }

```

Figure 16. Definition of the execution body of the main thread

图 16. Main 线程的执行体定义

```

void rt_system_scheduler_start(void)
{
    struct rt_thread *to_thread;
    rt_ubase_t highest_ready_priority;

    to_thread = _scheduler_get_highest_priority_thread(&highest_ready_priority);

#ifdef RT_USING_SMP
    to_thread->oncpu = rt_hw_cpu_id();
#else
    rt_current_thread = to_thread;
#endif /* RT_USING_SMP */

    rt_schedule_remove_thread(to_thread);
    to_thread->stat = RT_THREAD_RUNNING;

    /* switch to new thread */
#ifdef RT_USING_SMP
    rt_hw_context_switch_to((rt_ubase_t)&to_thread->sp, to_thread);
#else
    rt_hw_context_switch_to((rt_ubase_t)&to_thread->sp);
#endif /* RT_USING_SMP */

    /* never come back */
}

```

Figure 17. Code flow for starting the OS

图 17. 启动 OS 的代码流程

4. 结语

本文基于 QEMU 下的 vexpress-a9 平台，完整呈现了 RT-Thread 从加电复位到多任务调度的全链路启动与调试过程。通过“指令级仿真 + GDB 远程”的组合，我们无需任何硬件即可对 MMU 建立、页表配置、中断控制器初始化、堆管理、线程创建与调度等关键步骤进行源码级单步追踪，将传统“黑盒”启动流程转化为可视、可重复、可回溯的环境。使用该平台，我们对 RT-Thread 这一国产系统进行了源码级初始化流程分析，为我们快速掌握 RT-Thread 的开发提供了帮助。

结果表明，QEMU 能够 1:1 还原 Cortex-A9 多核 SoC 的行为，开发者可在 30 分钟内完成“裸机→操作系统→应用 main()”的完整镜像构建，显著缩短研发周期。

参考文献

- [1] QEMU Project. <https://www.qemu.org/>

-
- [2] RT-Thread. <https://www.rt-thread.org/>
 - [3] ARM® Architecture Reference Manual.
<https://documentation-service.arm.com/static/5f8dacff86e16515cdb8bb1?token=>
 - [4] 邱伟, 熊谱翔, 朱天龙. 嵌入式实时操作系统 RT-Thread 设计与实现[M]. 北京: 机械工业出版社, 2019.
 - [5] 郑苗秀. RT-Thread 实时操作系统内核、驱动和应用开发技术[M]. 北京: 电子工业出版社, 2019.