eCos 操作系统针对并行 S698P – SOC 的启动初始化过程研究

骆杰华 梁宝玉 颜 军
欧比特(珠海)软件工程有限公司, 珠海 519080

摘 要 实时操作系统 eCos 是一款嵌入式操作系统。eCos 支持多核并行处理系统,但是对多核处理器存在一些限制。多核处理器 S698P – SOC 满足 eCos 对处理器的限制要求。eCos 在 S698P – SOC 上的启动初始化过程包括主 CPU 的启动和从 CPU 的启动两部分。

关键词 eCos; 多核处理器; S698P – SOC
中图分类号: TP332 **文献标识码**: A
文章编号: 1006-3242(2010)04-0066-05

Initialization of eCos on the S698P-SOC

LUO Jiehua LIANG Baoyu YAN Jun
Orbita Software Engineering Inc., Zhuhai 519080, China

Abstract *Real time operating system* eCos *is an embedded operating system.* eCos *supports multi-core parallel processing system, but there are some restrictions.* S698P-SOC *(a multi-core processor) satisfies the restriction of* eCos. *The initialization of* eCos *on* S698P-SOC *includes two parts: the initialization of main-*CPU *and the initialization of sub-*CPU.

Key words eCos; *Multi-core*; S698P-SOC

对称多处理(Symmetrical Multi – Processing, SMP)技术,是指在一个计算机上汇集了一组 CPU,各 CPU 之间共享操作系统、内存子系统、总线结构和 I/O 系统等。在这种架构中,一台计算机不再由单个 CPU 组成,而同时由多个 CPU 运行操作系统的单一复本、共享内存和一台计算机的其它资源。系统将任务队列对称地分布于多个 CPU 之上,从而极大地提高了整个系统的数据处理能力。系统启动的时候,有主 CPU 和从 CPU 之分。在系统启动完成后,所有的 CPU 无主 CPU 和从 CPU 之分。所有的 CPU 都可以平等地访问内存、I/O 和外部中断。在对称多处理系统中,所有 CPU 共享系统资源,所有可用 CPU 能够均匀地分担工作任务。

1 S698P – SOC 处理器概述

S698P – SOC 处理器是一款高性能的、基于 SPARC V8 架构的、32 – bit RISC 嵌入式 SMP 微处理器。S698P – SOC 是高性能四核处理器,采用并行架构,每个 CPU 都可以独立运行。四个 CPU 通过 AMBA 总线互连,并共享存贮器和 I/O 等外设。图 1 是 S698P – SOC 处理器的内部结构图。

收稿日期: 2007-04-23
作者简介: 骆杰华(1982 –),男,广东人,本科,工程师,从事 S698P4 – SOC 四核并行处理器的测试与应用开发,eCos 操作系统针对 S698P4 – SOC 的应用开发;梁宝玉(1974 –),男,山东人,硕士,研究方向为高可靠 SOC 设计;颜 军(1962 –),男,山东高密人,博士,研究方向为智能控制、模糊控制、高可靠嵌入式控制器、SOC 芯片的设计及产业化。

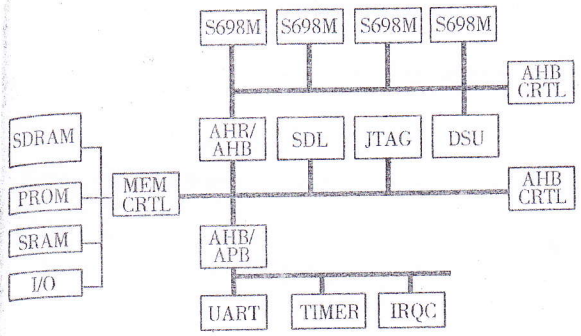


图1　S698P-SOC处理器内核结构

S698P-SOC各CPU之间的通信是通过多核中断控制器(MP IRQCTRL)的中断来实现的,其结构如图2所示。

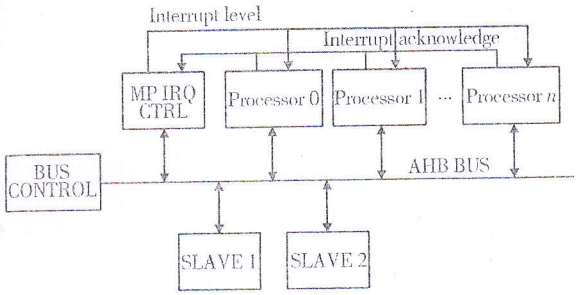
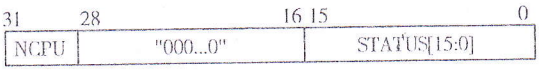


图2　S698P-SOC内部通信结构

S698P-SOC中的每个CPU都可以通过多核中断控制器向其它CPU发中断请求,每个CPU都可以响应其它CPU的中断请求。在多处理器状态寄存器(Multi-processor status register)的后16位(STATUS[15:0])写入1,相应的CPU启动。



2　eCos系统对处理器的要求

eCos(Embedded Configurable Operating System,嵌入式可配置操作系统)是一种针对16位、32位和64位处理器的可移植的开源嵌入式实时操作系统。eCos可在选定的处理器架构和平台上提供SMP支持。

eCos在它的SMP支持中加入了一些目标硬件限制,包括以下内容:

1)最大支持CPU个数为8,典型是2或4。

2)硬件必须提供"测试并设置"指令,或者"比较并交换"指令同步机制。eCos内核使用这些硬件指令实现螺旋锁。

3)不使用cache,或所有的CPU共享系统cache,或硬件维护cache的一致性。

4)共享内存的地址对所有CPU是一致的。

5)每个CPU都可以访问任何外设。

6)中断控制器必须将中断信号传送给指定的CPU。

7)允许一个CPU上的事件导致另一个CPU的重新调度(需要允许系统中一个CPU中断另一个CPU的机制)。

8)在某个CPU上运行的软件必须能识别其所运行的CPU。

S698P-SOC可以满足eCos对SMP的硬件限制条件:

1)S698P-SOC目前支持4个CPU。

2)S698P-SOC内有swapa指令,可以提供"测试并设置"同步机制,利用该条指令在eCos实现HAL_TAS_SET和HAL_TAS_CLEAR两个宏定义,eCos在HAL层以这两个宏来实现螺旋锁。

3)在SMP系统中,各CPU通过cache访问内存数据时,要求系统必须经常保持内存中的数据与cache中的数据一致。若cache的内容更新了,内存中的内容也应该相应地更新,否则就会影响系统数据的一致性。S698P-SOC用总线侦听技术(snooping)维护了4个CPU的data cache一致性。

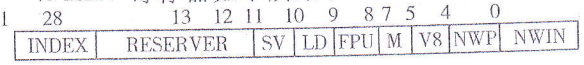
4)S698P-SOC所有的CPU共享系统SRAM和SDRAM,其编址是一致的,编程容易。

5)S698P-SOC所有外设的寄存器,都能被所有CPU访问到。所有外设的中断,都能引起所有CPU的中断处理,并且可以通过设置各CPU的中断屏蔽寄存器,指明哪个外设的中断由哪个CPU处理。

6)S698P-SOC各个CPU之间通过多核中断控制器(MP IRQCTRL)的中断来通信,允许将中断传递到系统中指定的CPU,该CPU执行中断处理函数,根据消息的内容执行优先级抢占重新调度或者时间片轮转调度。

7)S698P-SOC每个CPU均有一个称为%asr17的寄存器,其31-28位(下图INDEX部分)指明当前哪个CPU在运行,并由此实现CYG_KERNEL_CPU_THIS()宏定义,该宏定义返回当前运行的CPU号,程序可以根据这个宏定义作相应的处理。

%asr17寄存器如下所示:



3 启动初始化过程

SMP 启动时，CPU 有主 CPU 和从 CPU 之分。主 CPU 的启动顺序和从 CPU 的启动顺序是不同的。其中确定一个 CPU，称为主 CPU（在 S698P - SOC 中，编号为 0 的 CPU 是主 CPU）。主 CPU 处理启动初始化顺序。从 CPU，被 HAL 置于待机状态，或者被 HAL 置于空闲循环。当应用程序调用 cyg_sheduler_start 函数，从 CPU 开始启动运行。因此，S698P - SOC 处理器上的启动初始化过程必然会根据% asr17 寄存器区分主 CPU 和从 CPU，进行相应的初始化。

eCos 在 S698P - SOC 处理器上的启动初始化流程如图 3 所示。

下面详细说明图 3 所示的每个操作：

操作 1：eCos 上电或者软复位后，S698P - SOC 只启动 CPU0，其它 CPU 处于 power down 状态。系统的程序指针会自动指向地址 0。程序搬移到开始地址为 0x40000000 的内存区。运行的第一个程序是体系结构抽象层 arch 子目录中的 vector. S 程序。

操作 2：vector. S 一开始定义了 4K 字节的 trap_table，该表包括 256 组 trap。其中比较重要的有 0 号 trap（硬件复位中断向量）、5 号和 6 号 trap（寄存器窗口上溢和下溢中断向量）、17 - 31 号 trap（外设中断 1 - 15）、128 号 trap（操作系统软件软中断向量）。S698P - SOC 所有 CPU 的中断向量表是一样的。通过设置中断相关寄存器，指定一个 CPU 响应特定的外设中断，而其它 CPU 不响应中断。

操作 3：S698P - SOC 的 CPU0（其它 CPU 也一样）根据 0 号 trap，跳转到 genuine_reset。函数 genuine_reset 通过设置% tbr 寄存器，指定中断向量表的位置为 0x40000000 开始的 4K BYTE 区域。函数 genuine_reset 设置% wim 指明 CPU 的寄存器窗口数为 8，清空指令 cache 和数据 cache，设置总线侦听使能（snoop enable），使各个 CPU 的数据 cache 一致。

操作 4：S698P - SOC 所有 CPU 根据% asr17（处理器配置寄存器）的 31 - 28 位，得到一个 CPU 的索引号，指明当前哪个 CPU 在运行。执行代码如下：

```
rd   % asr17, % g2
srl  % g2, 28, % g2
```

% g2 为当前运行 CPU 的索引号，主 CPU 为 0，其它 CPU 为大于 0 的数字。

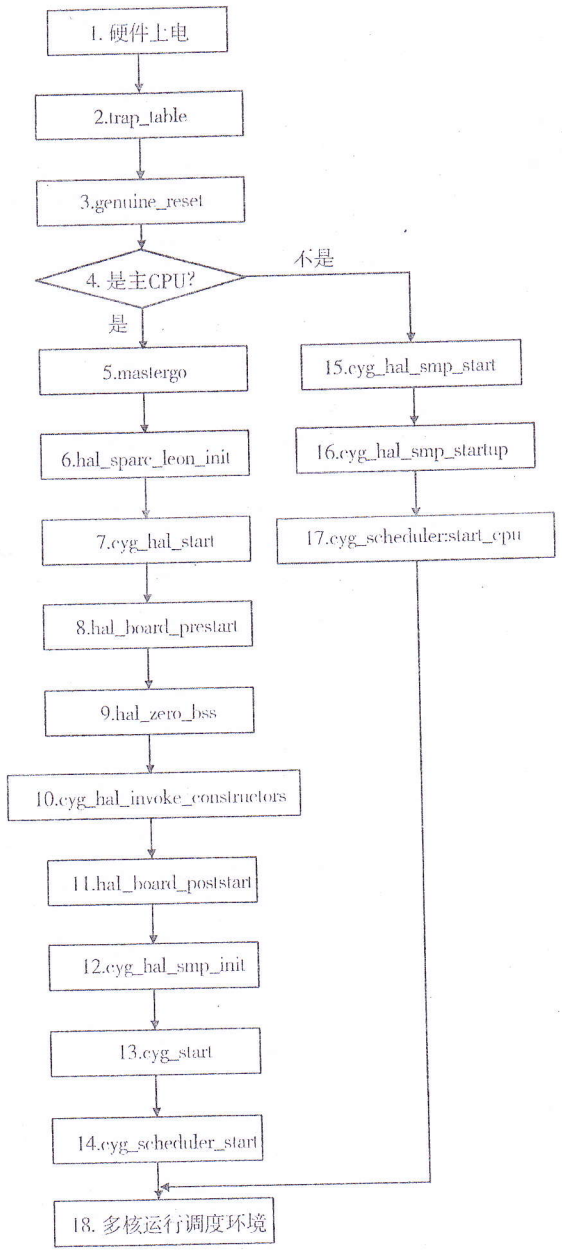


图 3 启动初始化过程

操作 5：如果是 CPU0，跳转到 mastergo。

操作 6：S698P - SOC 的 CPU0 调用 hal_sparc_leon_init 函数。该函数调用 ahbslv_scan 函数扫描 IP 核在 AHB 总线上的 slave 设备，调用 apbslv_scan 函数扫描 IP 核在 APB 总线上的 slave 设备。函数 hal_sparc_leon_init 初始化中断控制器、定时器、串口。函数 hal_sparc_leon_init 设置 CPU0 栈地址，% wim，% psr。函数 hal_sparc_leon_init 如果不使用 DSU，则设置内存配制寄存器 1、内存配制寄存器 2、sdram 控制寄存器。

操作 7：S698P - SOC 的 CPU0 调用 cyg_hal_start

函数,该函数执行S698P-SOC系统的硬件初始化操作。

操作8:S698P-SOC的CPU0在cyg_hal_start函数中调用hal_board_prestart函数,执行目标电路板开始运行前的处理动作。

操作9:S698P-SOC的CPU0在cyg_hal_start函数调用hal_zero_bss函数,清空bbs段。

操作10:接下来在cyg_hal_start函数调用cyg_hal_invoke_constructors函数,执行各个内嵌的构造函数。

操作11:cyg_hal_start函数调用hal_board_poststart函数,执行目标电路板开始运行后的处理动作。hal_board_poststart函数调用amba_init函数和开放系统中断。amba_init函数执行amba总线扫描,找出中断控制寄存器和定时器的起始地址。操作系统以后可以根据中断控制寄存器和定时器的起始地址对两者的寄存器进行配置。目前操作系统所需要的系统实时时钟是第一个定时器Timer1,中断编号为8。定时器的初始化函数如下:

```
void hal_sparc_leon3_clock_init(cyg_uint32 period) {
    cyg_hal_sparc_clock_period = (period);
    if (LEON3_GpTimer_Regs) {
    //设置定时器周期长度
    LEON3_BYPASS_STORE_PA(&LEON3_GpTimer_Regs->e[0].val,(period));
    LEON3_BYPASS_STORE_PA(&LEON3_GpTimer_Regs->e[0].rld,(period));
    LEON3_BYPASS_STORE_PA(&LEON3_GpTimer_Regs->e[0].ctrl,0);
    //设置定时器使能并自动重启动
    LEON3_BYPASS_STORE_PA(&LEON3_GpTimer_Regs->e[0].ctrl,
    LEON3_GPTIMER_EN |
    LEON3_GPTIMER_RL |
    LEON3_GPTIMER_IRQEN |
    LEON3_GPTIMER_LD);
    } else {
    diag_printf("Clock init failed");
    }
    }
```

操作12:S698P-SOC的CPU0调用cyg_hal_smp_init函数。函数cyg_hal_smp_init检查CPU0是否打开了总线侦听机制。函数cyg_hal_smp_init计算系统所有CPU的中断堆栈地址。函数cyg_hal_smp_init设置变量cyg_hal_smp_cpu_running第0位为1,表明CPU0已经启动。变量cyg_hal_smp_cpu_running_count加1,表示系统中启动的CPU数目加了一个。

检查打开总线侦听的代码如下:

```
if (cfg & ASI_LEON3_SYSCTRL_CFG_SNOOPING) {
    sparc_leon3_enable_snooping();
    }
    else {
    leon3smp_diag_printf("Caches disabled due to lack snooping\n");
    sparc_leon3_disable_cache();
    }
```

代码首先检查是否IP核配置了snooping,有则打开snooping,没有则报警并关闭data cache。

操作13:S698P-SOC的CPU0调用cyg_start函数,该函数是eCos的用户程序入口函数。用户程序至少要在这个函数内生成一个任务,并且这个任务处于可运行状态。

操作14:用户程序在cyg_start函数内调用cyg_scheduler_start函数,函数cyg_scheduler_start调用调度器类的start函数。在start函数内,CPU0调用cyg_hal_cpu_reset函数启动从CPU,并且调用调度器类Cyg_Scheduler的start_cpu函数。如操作17所示,CPU0也加载了CPU之间通信的中断处理函数,并执行第一个任务。

操作15:,在操作4中,如果准备启动的CPU是S698P-SOC的从CPU(从CPU是由CPU0在mp-status寄存器写入相应的位启动),程序执行函数cyg_hal_smp_start。函数cyg_hal_smp_start设置%tbr,设置中断堆栈,设置%psr。

操作16:S698P-SOC中的从CPU调用cyg_hal_smp_startup函数。函数cyg_hal_smp_startup设置该CPU已运行标志(置位变量cyg_hal_smp_cpu_running的相应位)。函数cyg_hal_smp_startup检查是否内核设置了总线侦听,否则打印告警信息。函数cyg_hal_smp_startup里,系统运行CPU数量加1,指明多启动了一个核,以后系统调度可以使用该核。

操作17:S698P-SOC的从CPU调用cyg_kernel_smp_startup函数。在该函数内先调用Cyg_

Scheduler::lock()锁调度器,再调用 Cyg_Scheduler::start_cpu()。函数 cyg_kernel_smp_startup 如下:

```
void Cyg_Scheduler::start_cpu()
{
  # ifdef CYGPKG_KERNEL_SMP_SUPPORT
Cyg_Interrupt * intr = new( (void *)&cyg_sched_cpu_interrupt[HAL_SMP_CPU_THIS()] )
Cyg_Interrupt( CYGNUM_HAL_SMP_CPU_INTERRUPT_VECTOR( HAL_SMP_CPU_THIS() ),
0,
0,
cyg_hal_cpu_message_isr,
cyg_hal_cpu_message_dsr
);
intr->set_cpu( intr->get_vector(), HAL_SMP_CPU_THIS() );
intr->attach();
intr->unmask_interrupt( intr->get_vector() );
  # endif
register Cyg_Thread * next = scheduler.schedule();
clear_need_reschedule(); // finished rescheduling
set_current_thread(next); // restore current thread pointer
  # ifdef CYGVAR_KERNEL_COUNTERS_CLOCK
CYG_REFERENCE_OBJECT( Cyg_Clock::real_time_clock );
  # endif
HAL_THREAD_LOAD_CONTEXT( &next->stack_ptr );
}
```

函数 cyg_kernel_smp_startup 一开始生成中断处理类 Cyg_Interrupt 的一个实例,设置该中断在指定的 CPU 处理。函数 cyg_kernel_smp_startup 调用 attach()函数。函数 attach()指定一个 CPU 处理特定的中断(目前为 14 号中断),并挂中断处理函数 isr 和 dsr。该函数开放指定的中断。函数 cyg_kernel_smp_startup 最后从调度器得到应该调度的一个任务,用 HAL_THREAD_LOAD_CONTEXT 载入该任务的上下文环境。从此,该 CPU 进入了多核运行调度环境。

操作 18:操作 17 完成后,多核初始化完成。系统进入多核运行环境。

总之,HAL 中不同的模块负责不同的初始化工作,如果对 HAL 配置选项的设置不同,启动过程也可能有异。主 CPU 执行操作 1->2->3->4->5->6->7->8->9->10->11->12->13->14 后,如果需要启动从 CPU,调用函数 cyg_scheduler_start 启动其它三个从 CPU。从 CPU 启动后,执行 1->2->3->4->15->16->17->18。两者都执行完成后,系统进入了多核运行调度环境。eCos 操作系统开始根据用户程序生成的任务以及任务之间的相互关系进行调度。至此,eCos 操作系统针对 S698P-SOC 的启动初始化过程结束。

4 结束语

本文主要研究了 eCos 操作系统在 S698P-SOC 的启动初始化过程。移植代码已在处理器上经过实际测试,系统稳定可靠,可运行多任务应用程序。本文所具体讨论的启动过程不仅有助于开发人员了解 eCos 操作系统的运行,而且有助于了解 S698P-SOC 多核处理器的工作原理,方便开发人员使用 S698P-SOC 多核处理器进行军用、民用领域项目开发。

参 考 文 献

[1] 欧比特(珠海)软件工程有限公司. S698P-SOC eCos Reference Manual[Z],2006.

[2] 欧比特(珠海)软件工程有限公司. eCos 使用说明[Z],2006.

[3] 欧比特(珠海)软件工程有限公司. 嵌入式操作系统 eCos 在 S698 系列处理器上的移植[Z],2006.

[4] Massa A J. 嵌入式可配置实时操作系统 eCos 软件开发[M]. 颜若麟,等,译. 北京:北京航空航天大学出版社,2006.