SmartDsp OS

海外开发一部　胡克俊

2009-8-4

SmartDSP OS 特点概述

- 运行于飞斯卡尔DSP上的基于优先级调度的实时操作系统.
- OS的大部分函数使用 ANSI C编写.
- 但是, 有时间严格要求的函数使用汇编语言作了优化，以利用Starcore 中可多语句并行执行的特点和硬循环的功能.

SmartDSP OS 启动流程

```
/*****************************************************************************/

void main()
{
    os_status status;    // The only local variable in main()

    /* OS Initialization - call before any other OS API calls. */
    status = osInitialize();
    if (status != OS_SUCCESS) OS_ASSERT;

    /* Interrupts are disabled until osStart() is called.
       You must not enable interrupts at this point !!! */

    /* Place any required initialization code within appInit().
       Using local variables inside main() is NOT recommended, because
       its stack is never restored - osStart() never returns !!! */
    status = appInit();
    if (status != OS_SUCCESS) OS_ASSERT;

    /* Start the OS with the background task. OS ticks now start.
       appBackground() should not return unless there is an error. */
    status = osStart(appBackground);
    if (status != OS_SUCCESS) OS_ASSERT;

    /* Execution reaches this point only if an error occurs. */
}

/*****************************************************************************/
```

SmartDSP OS 各模块

- 硬中断
- 软中断
- 多任务
- 软定时器
- 硬定时器
- 多核同步
- 核间消息
- DMA模块
- 网络模块
- MMU模块
- Cache模块

硬中断

- 中断处理器为中断处理函数作存储和恢复寄存器的工作.
- 共有31种不同的中断优先级
- 支持中断嵌套，高优先级中断可以打断低优先级中断的执行，待高优先级中断处理完成后继续进行低优先级中断处理

硬中断的禁止和激活

- 中断在osStart()被调用之前没有被激活
- 中断在运行时可以被enable和disable
  - osHwiSwiftDisable()
  - osHwiDisable()
  - osHwiSwiftEnable()
  - osHwiEnable(os_hwi_status prev_status)

创建硬中断处理函数

- 使用osHwiCreate可以将中断源与中断处理函数连接起来
- 参数是: 中断源, 中断优先级, 中断模式, 中断处理函数,中断处理函数的参数.
- status = osHwiCreate(OS_INT_UART,
  OS_HWI_PRIORITY0,
  LEVEL_MODE,
  uartGeneralIsr,
  0);

软中断

- 可以在OS的配置文件里配置软中断的个数.
- 软中断拥有一个叫count value的值，可用来向软中断处理函数传送信息
- 软中断处理函数执行时可被硬中断和更高优先级的软中断打断，在硬中断和更高优先级的软中断处理函数执行完之后继续执行

软中断

- 软中断可以被硬中断、其它软中断和task所触发.
- 软中断拥有16个不同的优先级.
- 软中断也可以被enable和disable

  os_swi_priority osSwiDisable();

  void osSwiEnable(os_swi_priority priority);

创建SWI

```
osSwiFind(&g_swi_holder1);
status = osSwiCreate(testTask1,
                     g_swi_holder1,
                     OS_SWI_PRIORITY0,
                     50);
if (status != OS_SUCCESS) OS_ASSERT;

osSwiCountSet(g_swi_holder1, 100);

status = osSwiActivate(g_swi_holder1);
if (status != OS_SUCCESS) OS_ASSERT;
```

Task

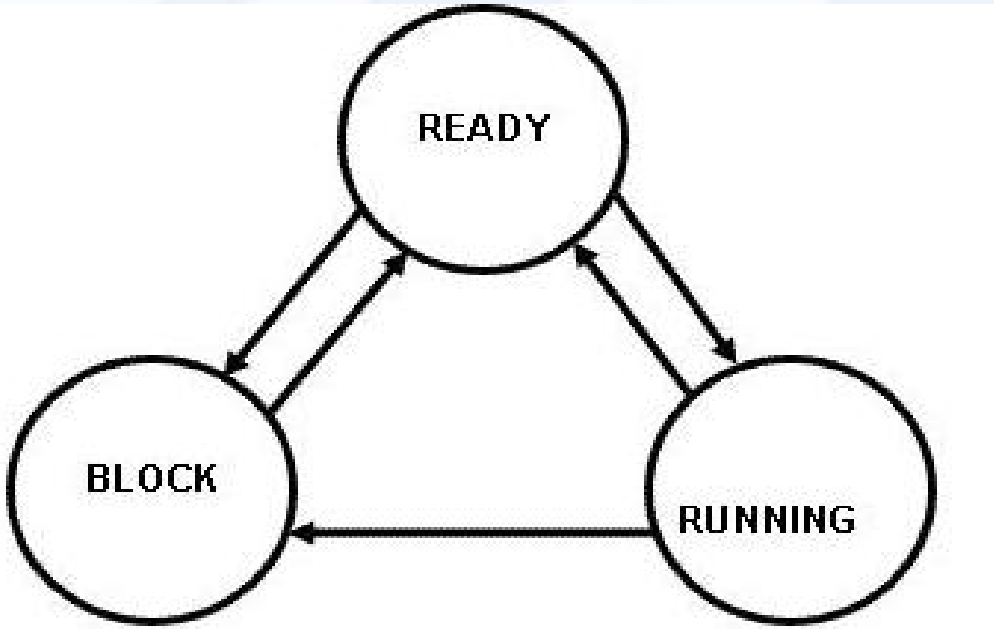
- 每一个task都拥有自己的优先级、栈区、task ID、名字等.
- 典型的task是一个无限循环，在五种状态之间切换，分别是unused, acquired, ready, running, blocked.
- Task的调度是基于优先级的，一般总是优先级最高的task在执行，除非调度器被锁定，或者某个软中断或硬中断被激活

Task

- Task拥有31个优先级
- 在相同优先级上可以有多个task
- 相同优先级的task之间可以转让CPU的执行权限，用户可以据此实现round robin调度或者其它的调度方式
- Background task是一种特别的task，它总是存在而且不能被挂起；因为CPU总是要执行一定的代码

Task的三种基本状态

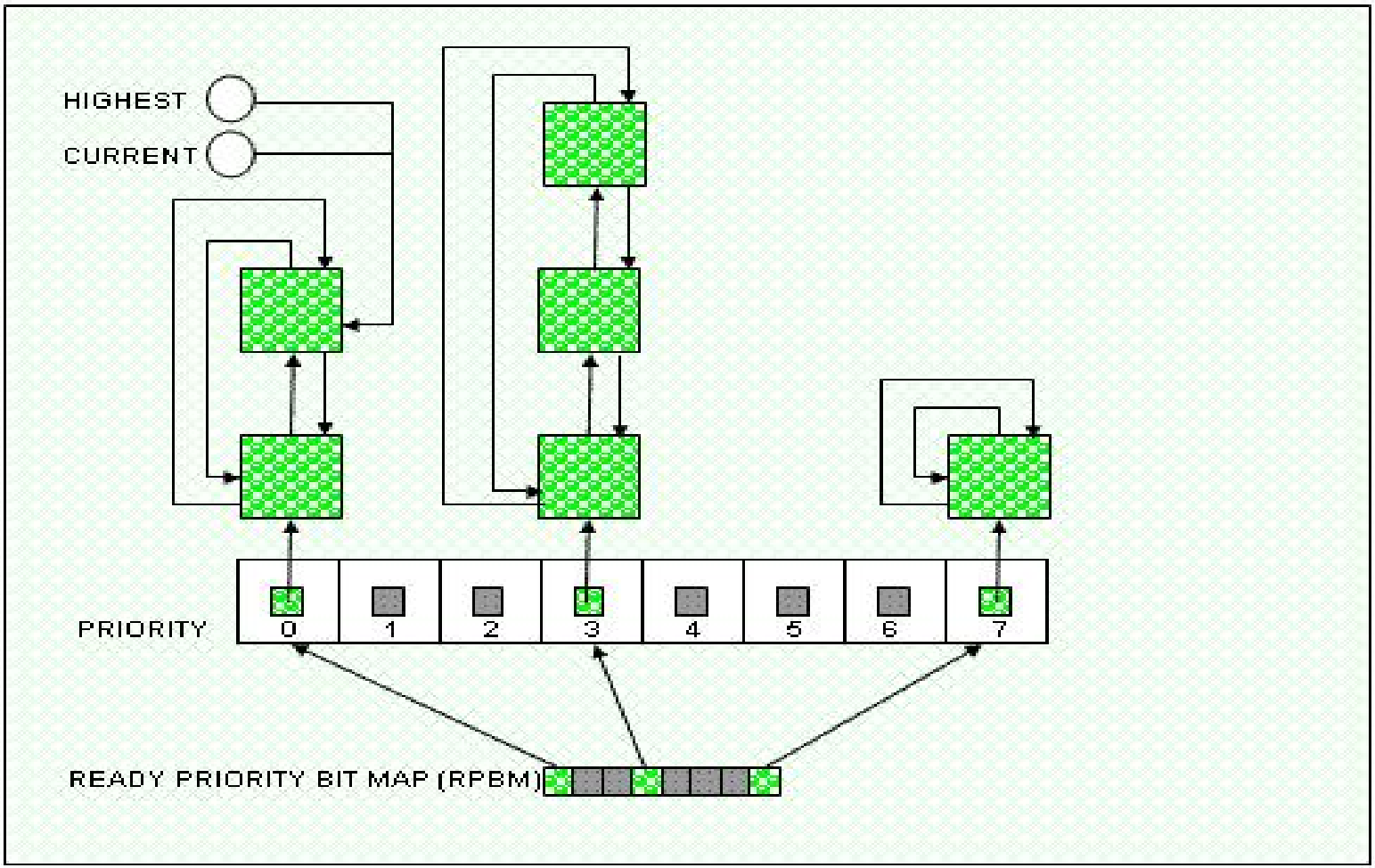
Task在三种基本的状态之间的切换



Task的状态切换

- Ready->Running:该task成为ready task中优先级最高的task
- Running->Ready:有更高优先级的task进入Ready状态
- Ready->Blocked:该task被其它task挂起
- Running->Blocked:该task挂起自己或调用了休眠函数
- Blocked->Ready:被挂起的task被其他task激活，或休眠时间到，或等待的事件发生

多任务调度



创建Task

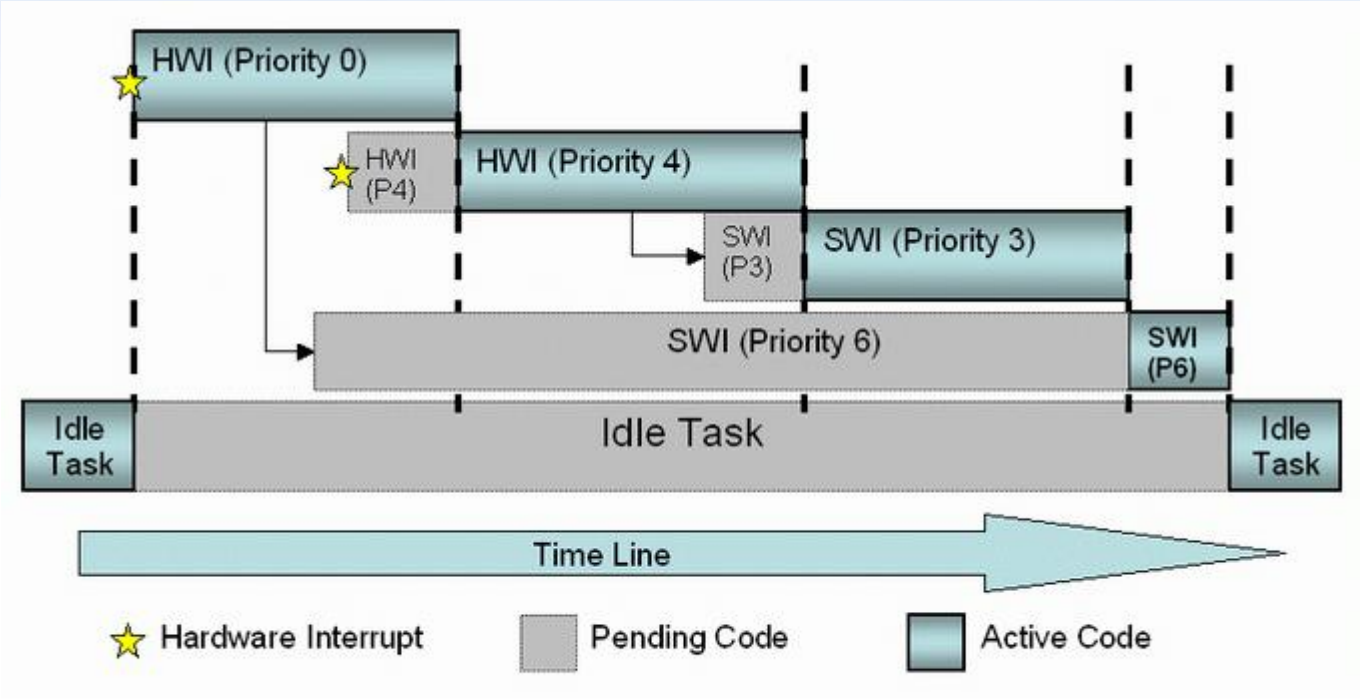
```
status = osTaskFind(&task_handle);
OS_ASSERT_COND(status == OS_SUCCESS);

os_task_init_param.task_function = F;
os_task_init_param.task_name     = N;
os_task_init_param.stack_size    = SZ;
os_task_init_param.task_arg      = A;
os_task_init_param.task          = task_handle;
os_task_init_param.task_priority = OS_TASK_PRIORITY_27;
os_task_init_param.private_data  = D2;
os_task_init_param.top_of_stack  = S;

status = osTaskCreate(&os_task_init_param);
OS_ASSERT_COND(status == OS_SUCCESS);
status = osTaskActivate(task_handle);
OS_ASSERT_COND(status == OS_SUCCESS);
```

调度器

- 每个core拥有独立的调度器
- Smart OS的任务调度是由事件驱动的，而不是采取时间片轮转的方法
- 各个任务的优先级如下所示

  - NMI
  - Hardware interrupts
  - Software interrupts
  - Tasks

HWI、SWI和Task之间的优先级关系



任务间的通信-Semaphore

- 可以使用Semaphore同步两个Task、或Task和SWI、或Task和HWI.
- HWI和SWI等待事件发生的时候，只能使用非阻塞函数osEventSemaphoreAccept().
- Background Task不能阻塞在Semaphore上，因为它不能被阻塞执行.
- Semaphore内拥有一个计数器，可以记录共享资源的个数

任务间的通信-Semaphore

- Semaphores的创建
  - osEventSemaphoreFind()
  - osEventSemaphoreCreate()
- Semaphores的使用
  - osEventSemaphorePend()
  - osEventSemaphoreAccept()
  - osEventSemaphorePost()
  - osEventSemaphoreReset()
- Semaphores的删除
  - osEventSemaphorePend()

任务间的通信-Event Queues

- Event Queues可以传送32bit的消息.
- Event Queues可用来在Task、HWI和SWI之间进行通信.
- HWI和SWI接收消息的时候，只能使用非阻塞函数osEventQueueAccept().
- Background Task不能阻塞在Event Queues上，因为它不能被阻塞执行.

任务间的通信-Event Queues

- Event Queues的创建
  - osEventQueueFind()
  - osEventQueueCreate()
- Event Queues的使用
  - osEventQueuePend()
  - osEventQueueAccept()
  - osEventQueuePost()
  - osEventQueueReset()

软定时器

- 软定时器使用硬件tick timer作为参考时间
  - 提高tick timer的频率可以提高软定时器的精确性
  - 降低tick timer的频率可以降低定时中断对core资源的消耗
- 所有的软定时器处理函数都可以视为一个SWI处理函数.
- 软定时器分为一次性定时器和周期性定时器.
- 可以在OS的配置文件中定义软定时器的最大数量

创建软定时器

```
os_timer_handle timer1;

osTimerFind(&timer1);
status = osTimerCreate(timer1,            // timer object
                       OS_TIMER_ONE_SHOT,  // timer mode
                       5,                  // ticks timeout
                       timerTest1);        // timer handler
if (status != OS_SUCCESS) OS_ASSERT;

// Start the timer
status = osTimerStart(timer1);
if (status != OS_SUCCESS) OS_ASSERT;

void timerTest1()
{
   // Do something...
}
```

硬定时器

- 硬定时器依赖于底层硬件并以系统频率进行工作
- 硬定时器在各个core之间共享
- 对硬定时器的使用和软定时器的使用相似，除了需要指定时钟来源
- 硬定时器与软定时器相比，定时时间更加精确，因为软定时器基于OS_TICK，OS_TICK的最高精度只有1ms

创建硬定时器

```
/* Create one-millisecond timer */
   status = osHwTimerFindForce(OS_HW_TIMER1);
   if (status != OS_SUCCESS) OS_ASSERT;
   if (osHwTimerCreate(OS_HW_TIMER1,
                  OS_TIMER_PERIODIC,
                  osGetSystemClock() * 2  * MHZ / (128 * 1000),
                  PRESCALER128SOURCE,
                  NULL,
                  OS_HWI_PRIORITY0) != OS_SUCCESS)
         OS_ASSERT;
```

核间同步

- 核间同步机制用来保护核间的共享资源. SmartDSP 提供了 spinlocks and barriers 两种方法.
- Spinlocks 的机制是用原子操作的方法查询共享变量以达到保护共享资源的办法.
- Barrier 用来作核间代码的同步，所有核的代码都调用的Barrier的同步函数后，才会继续向下执行.

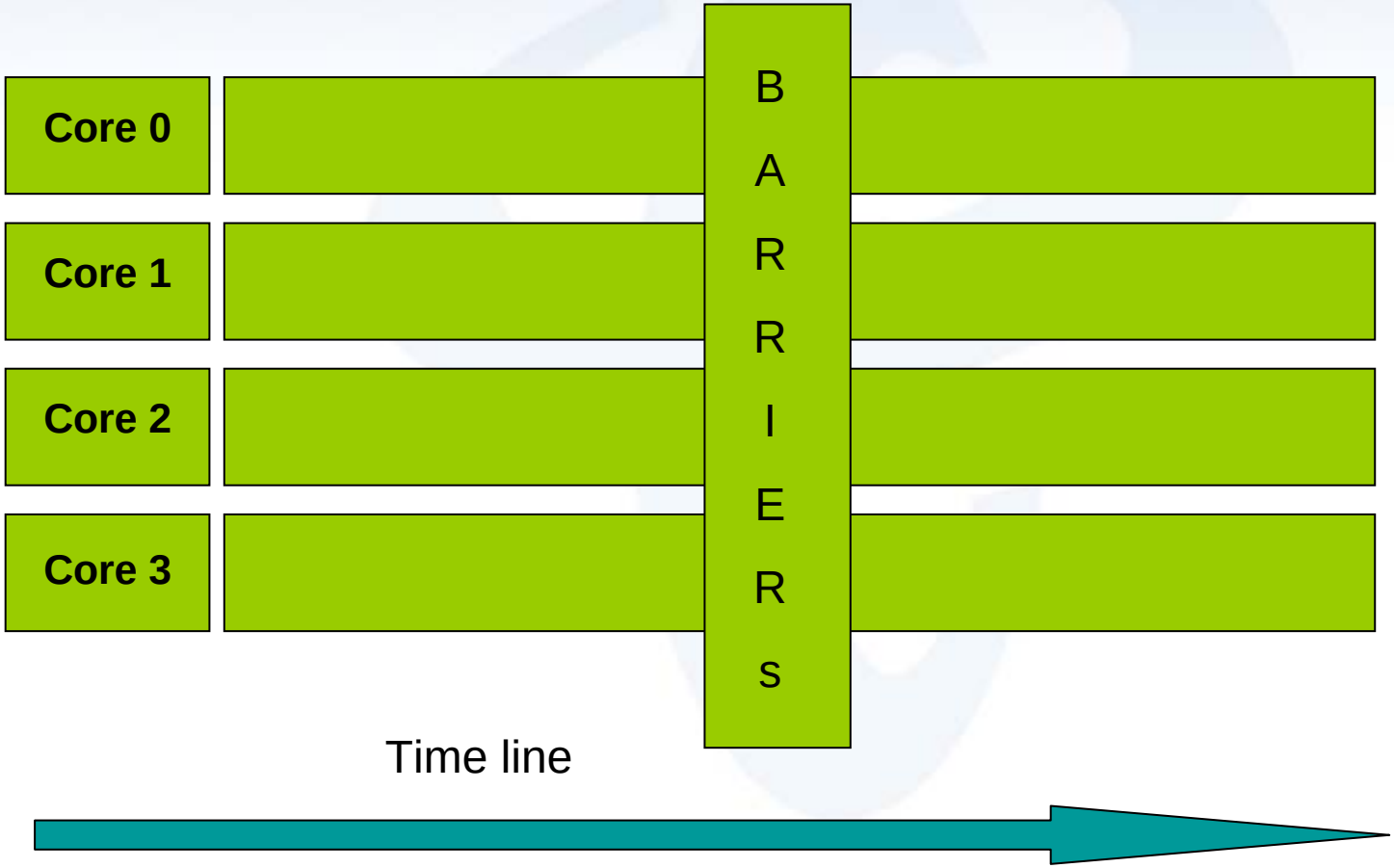
Spinlocks

- Spinlocks是在core之间保护关键数据和代码的一种方法
- 而在同一core上使用中断禁止的方法保护关键数据和代码
- Spinlock的实现基于原子操作bmtset.w
- 但是该原子操作仅在M2(MSC814x)或M3(MSC815x)区域上可行，并且不能使用cache
- 所有的Spinlock的API函数如下图所示

- osSpinLockGet()
  Spin until lock is taken. Multicore protection
- osSpinLockIrqGet()
  Disable interrupts and spin until lock is taken. Multicore, multitask protection
- osSpinLockTryGet()
  Try to get the spinlock. Returns OS_SUCCESS or OS_FAIL
- osSpinLockRelease()
  Release the spinlock
- osSpinLockIrqRelease()
  Release the spinlock and enable interrupts

核间同步-Barriers

Barriers 使所有的core进行同步



核间消息

- 在多核的条件下，运行在不同核上的应用程序相互通信的一种办法就是Inter-core消息
- Inter-core消息分为Intercore Messages和Intercore Message Queues.
- Inter-core消息基于虚拟中断(VIRQ (MSC814x)/interrupt mesh (MSC815 x))，而虚拟中断的个数是有限的，所以Inter-core消息的数量也是有限的

核间消息

- Intercore Messages可以指定向某一个核传送消息，Intercore Message Queues只能在所有核间进行消息广播
- Intercore Messages一次只能传送32bit的信息
- Intercore Message Queues一次可以传送32*Nbit的信息
- Intercore Messages要等待对方将消息取走才可以发送下一条信息， Intercore Message Queues没有这个限制

核间消息的两种方式

- 同步式：
  1.发送core获取mailbox的spinlock；
  2.发送core将数据写入mailbox；
  3.发送core发送中断通知受信core；
  4.受信core收到中断请求，请求中包含mailbox号；
  5.受信core读取数据
  6.受信core释放mailbox的spinlock

核间消息的两种方式

- 异步式：
  1.数据在初始化时定义好，作为中断处理函数的参数；
  2.送信core以中断通知受信core；
  3.受信core收到中断通知后，中断处理函数被激活，从输入参数获得信息

Intercore Messages

- Intercore Messages的API函数如下所示

osMessageFind()	- Find an available message
osMessageCreate()	- Create the message
osMessageCreateAsync()	- Create the message, data predefined
osMessagePost()	- Post the message
osMessagePostIrq()	- Disable interrupts and post the message
osMessagePostAsync()	- Create VIRQ, data predefined
osMessageGet()	- Get the message

Intercore Message Queues

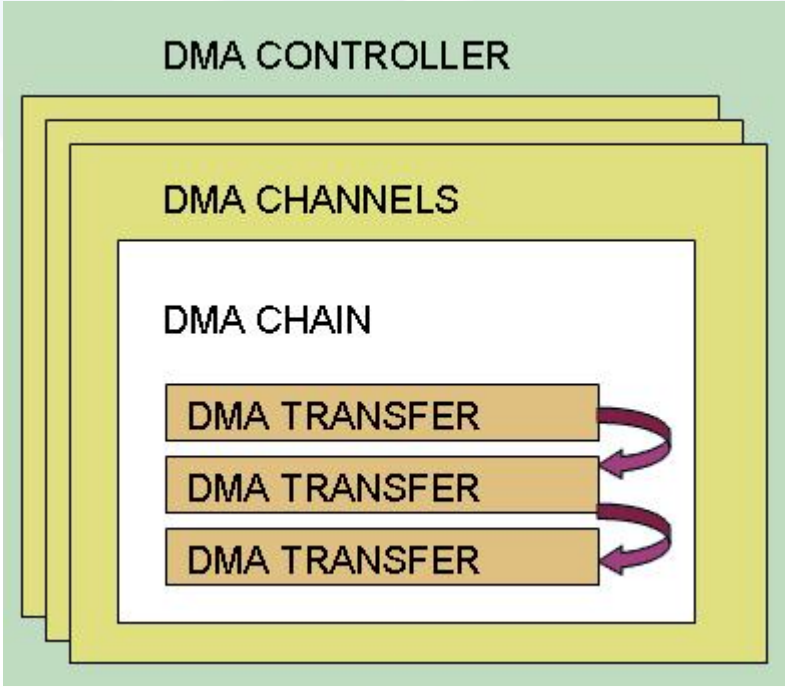
- Intercore Message Queues的API函数如下所示

osMessageQueueCreate() – Creates the queue. Message size defined by user. Queue number shared by all – no find.

osMessageQueuePost() – Post to queue.
osMessageQueueGet() – Get from queue.
osMessageQueueDispatcher() – Should be called from message queue handler. Calls osMessageQueueGet()

DMA模块

- Smart OS提供了统一的API函数，可以操纵各个平台上的DMA设备.
- 下图是DMA相关的概念

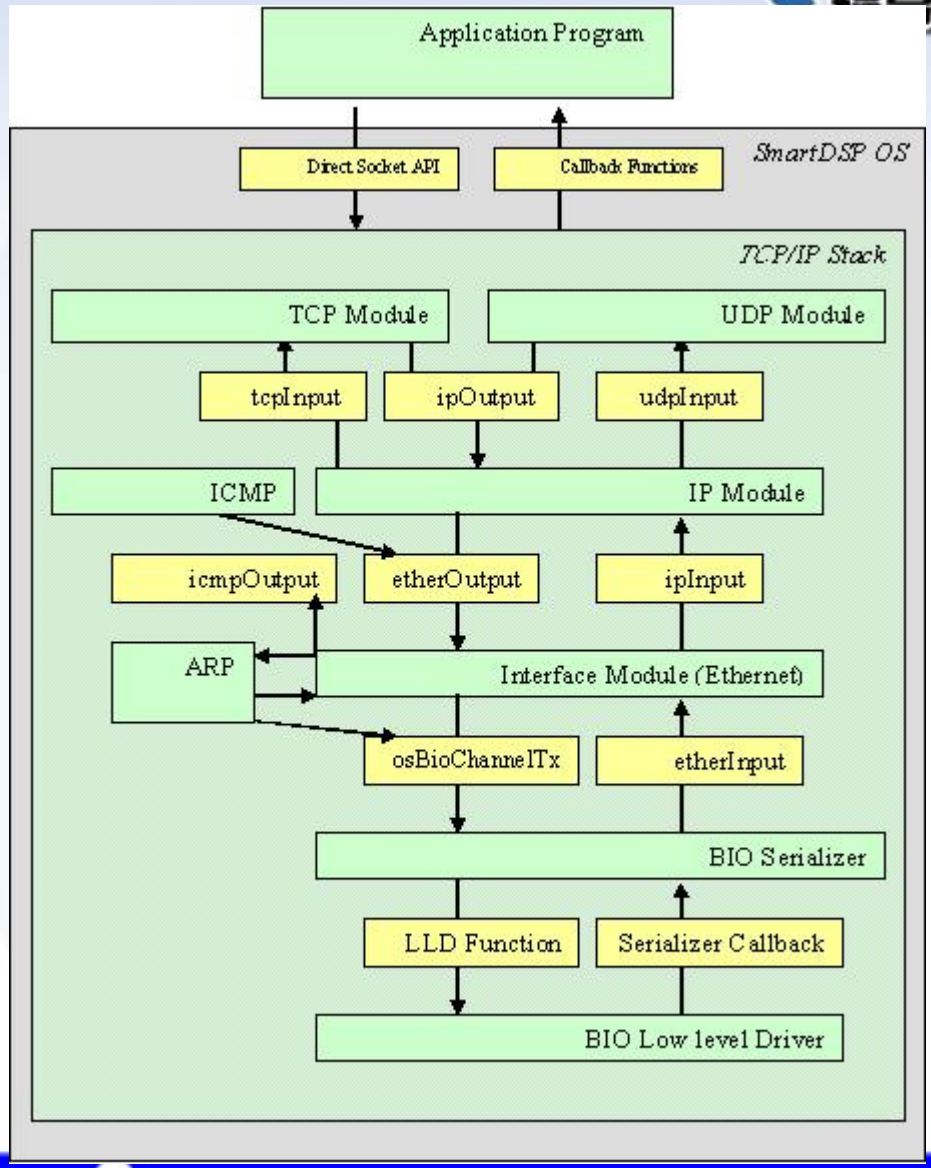


DMA模块

- DMA controller 管理所有的模块组件.
- DMA channel是执行DMA传送的单位.
- DMA chain是DMA transfers的容器.
- DMA transfers设置具体的传送源地址，目的地址等信息.
- DMA传送可以并行执行.

网络模块

- NET模块实现了TCP, UDP, IP, ICMP, ARP, DHCP and RTP等通信协议.

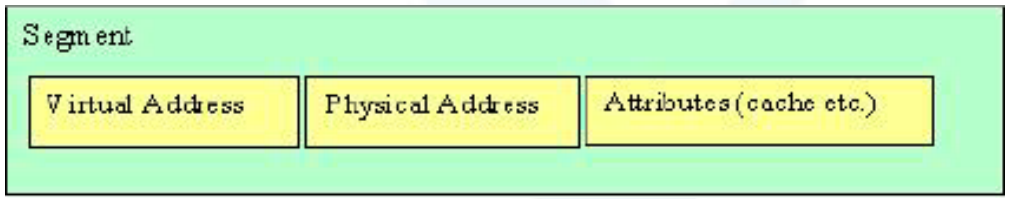


MMU模块

- 对代码和数据访问内存提供了硬件保护机制.
- 支持两种执行优先级.
- 支持255个Task ID.
- 可以高速的执行虚拟地址到物理地址的转换.
- MMU模块包含MMU segment和MMU Context两个概念

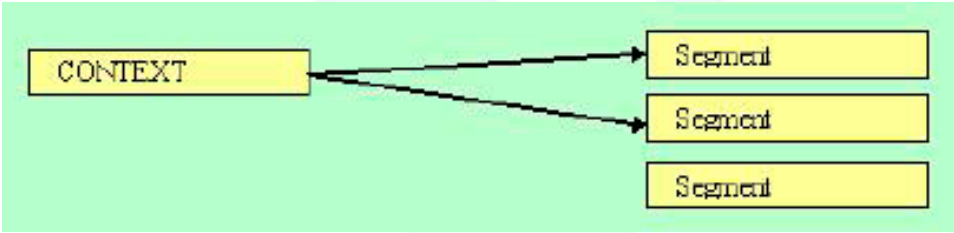
MMU segment

- 对内存区域设置读写权限
- 定义虚拟地址和物理地址的对应关系
- 定义内存段的cache机制(cache或non cache)
- MMU segment可以在LCF中预先指定，也可以在程序中创建



MMU Context

- 是一组segments的集合
- 每个MMU Context拥有一个唯一的ID号
- 存在两种MMU Context：程序Context和数据Context
- MMU Context可以在LCF中创建，也可以在程序中动态创建



Caches

- 是临时存储数据的高速缓存
- Cache包括L1 Cache和L2 Cache，前者容量小但速度快
- 将Cache和内存进行同步的命令Cache Sweep Commands(invalidate/flush/synchronize)

Caches

- Cache操作时必须对齐到cache line
- 为方便操作，Smart OS提供了下列两个宏

```
#define CACHE_OP_LOW_ADDR (VIRT_BASE, GRANULARITY)
#define CACHE_OP_HIGH_ADDR (VIRT_BASE, SIZE, GRANULARITY)
```

```
long read_from_maple(void* addr, void* data, uint32_t size, uint32_t dev_id)
{
    os_status           status;
    uint32_t            cache_addr_lo,cache_addr_hi;

    cache_addr_lo = CACHE_OP_LOW_ADDR(data, ARCH_CACHE_LINE_SIZE);
    cache_addr_hi = CACHE_OP_HIGH_ADDR(data, size, ARCH_CACHE_LINE_SIZE);

    osCacheDataSweep(
            (void*)cache_addr_lo,
            (void*)cache_addr_hi,
            MMU_SHARED_PID,
            CACHE_FLUSH);


    return (long)size;

}
```

谢谢！