REX OS 参考手册

****

文件名称：REX OS 参考手册
文件标识：ver1.0
摘 要：介绍从软件分析到软件开发完成的各个阶段
作 者：Cat King
完成日期：2004 年 10 月 4 日
* *
取代版本：Ver1.0
原作者 ：Cat King
完成日期：2005 年 08 月 11 日

****

Preface

Declaration:

In general I will freely answer any questions that I receive by email, or point you in the direction of a resource that may be of assistance.

At the moment I am busy with a couple of large ongoing projects and don't have the time to work on custom examples or small software projects. I would however be willing to entertain job offers ☺

Read the whole thing! If you have a question during one section of the tutorial just have a little patience and it might just be answered later on. If you just can't stand the thought of not knowing, at least skim or search (yes computers can do that) the rest of the document before asking the nice folks on IRC or by email.

Another thing to remember is that a question you might have about subject A might end up being answered in a discussion of B or C, or maybe L. So just look around a little.

Fell free to contact me (catking.wang@gmail.com)

Ok I think that's all the ranting I have to do for the moment, let's start my working.

Contents

REX 初始化

函数 main() 的开始就是调用 rex_init() 进行初始化工作。Rex_init() 初始化数据结构，启动空闲任务和一个单一任务，用户将该单一任务指定为参数。该任务是用户应用程序的入口；它负责启动后续的用户任务。

Rex_init()

Purpose

REX 的入口。它启动 REX 和应用程序代码。

Parameters

```
Void rex_init(
        Unsigned long       p_istack[],
        Unsigned long       p_istksiz[],
        Rex_tcb_type        *p_tcb,
        Unsigned long       p_stack[],
        Unsigned long       p_stksiz[],
        Unsigned long       p_pri[],
        Void                (*p_task) (unsigned long),
        Unsigned long       p_param
)
```

Parameter	参数使用
P_istack	REX 把这个值指定为全局变量，它表明中断所使用的栈的位置。
P_istksiz	REX 没用。
P_tcb	主控任务的 TCB。REX 使用其作为 rex_def_task()的一个参数。这个 TCB 是应用程序代码的入口点。
P_stack	Rex_def_task()的参数
P_stksiz	Rex_def_task()的参数
P_pri	Rex_def_task()的参数
P_task	Rex_def_task()的参数
P_param	Rex_def_task()的参数

基本算法：

1． Lock interrupts
2． 使用提供的 TCB，调用函数 rex_def_task() 启动主控任务。
3． Unlock interrupts

REX 任务的工作程序

每个 REX 任务都独立地由 REX 内核调度。每个任务都有一个数据结构 rex_tcb_struct 。
TCBs 按优先级顺序编排在一个单一的链接表中-------活动的和挂起的任务在同一个列表中。
每个任务将一直执行到下列事件之一发生而进行上下文切换时：

- 自愿挂起，比如执行了 rex_wait() or rex_timed_wait()
  rex_wait() 将遍历 TCB 列表，找到最高优先级的就绪任务，并告诉调度程序激活它。

- 在高优先级的挂起任务里设置了一个信号
  如果优先级为 30 的任务 A 被阻塞而等待信号 1，而优先级为 10 的任务 B 给任务 A 发送了该信号，那么任务 B 将挂起而由任务 A 替代执行。

- 产生一个更高优先级的任务。

如果优先级为 30 的任务 A 创建了一个新任务 C，其优先级为 50，那么任务 A 将挂起转而执行任务 C。

- 一个正执行的任务使另外一个任务的优先级比它自己的优先级高

如果优先级为 20 的任务 A 使得任务 B 的优先级从 10 升高到 30，那么任务 B 的上下文将切换进来，替代任务 A。

- 中断发生并且 ISR 切换进优先级比被中断的任务的优先级高的任务中。
  一旦从中断返回，控制权将交给优先级高的任务。（注意这种情况包括计时器期满事件，因为它是一个处理计时器倒计时的 ISR。）
  调度的其他细微差别如下：
- 如果中断例程在当前中断的任务中设置了一个信号，不发生上下文切换。而是在 ISR 之后接着执行当前的任务。
- 如果一个任务在它自己的 TCB 中设置了一个信号，不发生上下文切换，因为按照定义，当前任务仍然是优先级最高的就绪任务。
- 如果一个任务在低优先级的任务中设置了一个信号使得低优先级的任务就绪，不发生上下文切换。当前的任务将继续执行。只有当所有的优先级高的任务挂起时才会执行低优先级的任务。

REX 优先级的情况

REX 允许任务的优先级范围为 1 到 65535，1 是最低优先级。优先级 0 预留给空闲任务使用（REX 内部使用），它是优先级最低的任务。注意 REX 当前是使用一个 32-bit 的整数做为优先级，它允许的优先级大于 65535。但是，REX 应用程序坚持着 65535 这个限制，在将来完全可能使用 16-bit 的整数做为优先级以强化这个限制。因此，超过这个限制将冒跟将来的 REX 版本不兼容的风险，是极不鼓励的。

没有两个及两个以上的 REX 任务有相同的优先级。

相关的数据结构

TCB 是任务和调度的主要数据结构。它也是信号的主要数据结构；以下是当前 TCB 类型的一个简单描述。

```
Typedef struct rex_tcb_struct
{
    void            *sp;
    void            *stack_limit;
    word             slices;
    rex_sigs_type    sigs;
    tex_sigs_type    wait;
    word             pri;
 struct
    {
      struct    rex_tcb_struct *next_ptr;
      struct    rex_tcb_struct *prev_ptr;
    }link;
}rex_tcb_type;
```

Field	How REX uses field
Sp & stack_limit	Sp 指向任务栈，每次上下文切换时 REX 需要这个。Stack_limit 用作栈溢出检测特性。
Slices	记录该任务已经被调度的次数。对调试有用。
Sigs	该任务的当前设置信号。同大多数 OS 一样，信号不排队。它只是一个位掩模，因此一次性设置 10 次而没有读过同设置一次的效果是一样的。
Wait	任务正等待的信号的设置。如果这个非零，任务就挂起直到至少一个信号被设置。
Pri	任务优先级
link	将 TCB 链接起来，以便调度时遍历。

任务 API 函数

rex_def_task

Purpose

创建一个新的任务并将其置于可以调度的状态。

Parameters

```
Void rex_def_task(
    Rex_tcb_type        *p_tcb,
    Unsigned long       p_stack[],
    Unsigned long       p_stksiz,
    Unsigned long       p_pri,
    Void                (*p_task) (unsigned long),
    Unsigned long       p_param
)
```

Argument	How REX uses argument
*p_tcb	这是一个未初始化的 TCB 模块指针，它已经由调用函数分配了空间。REX 使用这个来存储跟任务相关的内容，包括它的上下文和运行条件。
P_stack[]	不再使用
P_stksiz	该值通知 REX 组成栈所需要的 16-bit 字的数量。
P_pri	任务优先级
*p_task	用户函数指针。REX 使用这个启动任务，但不必存储因为任务的入口只用来启动任务从不做为后面的参考。
P_param	REX 调用用户任务函数要带有这个参数。REX 不会检查一个参数－－只是简单地传递到新任务中。

Rex_self()

Purpose

返回当前正调用地任务（也就是，当前正执行的任务）的 TCB 的指针。

Parameters

```
Rex_tcb_type *rex_self(void)
```

优先级 API 函数

优先级 API 有三个函数：rex_set_pri(), rex_task_pri(), 和 rex_get_pri()。

Rex_set_pri()

Purpose

改变当前任务地优先级到一个新的值。

Parameters

```
Unsigned long rex_set_pri(
        Unsigned long p_pri
)
```

Parameter	How REX uses parameter
P_pri	REX 传递该值给调用 rex_task_pri()

Return value	What REX does
Unsigned long	REX 返回 rex_task_pri() 调用的结果。

Rex_task_pri()

Purpose

改变任何任务（当前任务或其他任务）的优先级为一个新值。

Parameters

```
Unsigned long rex_task_pri(
        Rex_tcb_type        *p_tcb,
        Unsigned long        p_pri
)
```

Parameter	How REX uses parameter
P_tcb	REX 在这个 TCB 中改变优先级
P_pri	REX 在这个 TCB 中设置该优先级

Return value	What REX does
Unsigned long	REX 返回改变之前生效的优先级

Rex_get_pri()

Purpose

返回当前任务的优先级。

Parameters

```
Unsigned long Rex_get_pri(void)
```

Return value	What REX does
Unsigned long	REX 从 TCB 中返回当前任务的优先级

REX 信号的工作程序

REX 的信号功能主要集中在 TCB 的两个 field：

- sigs　　是一个掩码，包括给定任务已经设置的信号。
- wait　　是一个掩码，包括一些信号，一旦这些信号被设置，任务就挂起。

只有 wait 掩码被清除任务才能调度。具体说来，就是，无论何时 REX 遍历 TCB 列表搜索待运行的任务，它将跳过有非零 wait 掩码的任务。

信号 API 有四个函数组成：rex_wait(), rex_set_sigs(), rex_get_sigs(), 和 rex_clr_sigs()。

Rex_wait()

Purpose

挂起当前的任务直到一组信号中的一个被设置。

Parameters

```
Unsigned long rex_wait(
    Rex_sigs_type     p_sigs
)
```

parameter	How REX uses parameter
P_sigs	REX 取走这个参数并执行或运算将其加到 TCB.wait 中。然后 REX 找优先级最高的就绪任务并调用调度程序将那个任务切换进来，替代调用 rex_wait() 的任务。

Return value	What REX does
Rex_sigs_type	Rex_wait() 要么： ■ 如果 wait 掩码信号已经设置立即返回 ■ 当信号最后到达时解除挂起后返回。 在两种情况下，当前的 set 信号被返回。

基本算法：

1． Lock interrupts
2． 比较 TCB.sigs 和提供的 wait 掩码。
3． 如果有匹配，设置 retVal 等于 TCB.sigs, unlock interrupts，并返回 retVal。
4． 如果没有匹配，将 wait 掩码传递到 TCB.Wait。
5． 挂起任务，调用另外一个任务。
6． 解除挂起，Copy TCB.sigs 到参数 retVal. Unlock interrupts， 返回 retVal.

Rex_set_sigs()

Purpose

Rex_set_sigs()取信号掩码做为参数并在目标任务的 TCB.sigs field 中设置这个信号。

Parameters

```
Rex_set_sigs(
    Rex_tcb_type        *p_tcb,
    Rex_sigs_type        p_sigs
)
```

Parameter	How REX uses parameter
*p_tcb	需要设置信号的 TCB 的指针。 REX 在 TCB.sigs 里面设置请求的信号。另外，REX 检测是否等待信号的时候任务已挂起。如果是，REX 清除该任务的 TCB.wait，使其成为就绪任务。如果该任务的优先级比当前任务高，将立即进行上下文切换。

Return value	What REX does
Rex_sigs_type	REX 在将新的信号加到掩码之前先将 TCB.sigs 拷贝到 prev_sigs. 返回时，它返回 prev_sigs.

基本算法

1. Lock interrupts
2. 捕获 TCB.sigs 到暂时存储器 prevSigs 变量。
3. 将新的信号加到 TCB.sigs。
4. 比较 TCB.sigs 和 TCB.wait 。如果有匹配，设置 TCB.wait 到 0。
5. Unlock interrupts.
6. 返回 prevSigs.

Rex_get_sigs()

Purpose

简单地返回 TCB 的 当前的 set 信号。

Parameters

```
Rex_sigs_type rex_get_sigs(
        Rex_tcb_type    *p_tcb
)
```

Parameter	How REX uses parameter
*p_tcb	需要返回信号的任务的 TCB 指针

Return value	What REX does
Rex_sigs_type	REX 返回 TCB.sigs

基本算法：

1． Lock interrupts
2． Copy TCB.sigs 到暂时变量 currSigs 中。
3． Unlock interrupts
4． 返回 currSigs

Rex_clr_sigs()

Purpose

清除指定的 TCB 请求的信号，在清除操作执行之前返回 TCB.sigs 的值。

Parameters

```
Rex_sigs_type rex_clr_sigs(
        Rex_tcb_type        *p_tcb,
        Rex_sigs_type        p_sigs
)
```

Parameter	How REX uses parameter
*p_tcb	需要清除信号的任务的 TCB 指针
P_sigs	这个包含了要清除的信号。REX 简单地对掩码取反 并同 TCB.sigs 进行与运算。

Return value	What REX does
Rex_sigs_type	在执行清除前，REX 备份 TCB.sigs 的原始状态，在清除完成后返回其给 caller。

基本算法：

1. Lock interrupts
2. Copy TCB.sigs into temporary variable *prevSigs*
3. clear *p_sigs* from TCB.sigs
4. Unlock interrupts
5. Return *prevSigs*.

REX timers

REX timers 如何工作

活动的 REX 计时器是一个具有 rex_timer_struct 结构的双向链接的列表。使用 REX 计时器的任务先得分配一个 rex_timer_struct, 然后调用 rex_def_timer(). Rex_def_timer() 装载这个结构并带有参数 TCB——期满时通知的对象和设置的信号——要发送给 TCB 的信号。 这时计时器并没有激活。

Rex_set_timer(), 被调用，设置计时器的初始计数，并将计时器插入到活动的计时器列表，意味着计时器将开始递减。

从硬件角度，当计时器芯片发送一个计时器滴答中断，CPU 就调用 ISR，ISR 调用 rex_tick()。Rex_tick()遍历计时器列表，每个计时器结构计数减 1，并通知哪个计时器期满（计数为 0 时表示期满）。对于期满的计时器，rex_tick() 查询计时器结构，找回要通知的任务的 TCB 和要设置的特殊信号，并执行它。计时器结构从活动列表中移走，直到再次调用 rex_set_timer(),计时器将再次成为列表的一部分。

相关的数据结构

```
typedef struct rex_timer_struct
{
        unsigned long               cnt;
        rex_tcb_type                *tcb_ptr;
        rex_sigs_type               sigs;
        struct
        {
                struct rex_timer_struct         *next_ptr;
                struct rex_timer_struct         *prev_ptr;
        }link;
}rex_timer_type;
```

Field	How REX uses field
Cnt	REX 用 rex_set_timer()设置该值。每次计时器 ISR 调用 rex_tick, cnt 就减 1。 如果 cnt 为 0，REX 就执行计时器期满操作。
*Tcb_ptr	Rex_def_timer()设置该值。当计时器期满时，REX 使用这个 TCB 做为信号设置的目标。
Sigs	当计时器期满时，在目标 TCB 中设置这个信号。
Link	REX 使用这个维护计时器列表，rex_tick()可以遍历列表。

Timer API 由 6 个函数组成：rex_tick(), rex_def_timer(), rex_set_timer(), rex_get_timer(), rex_clr_timer(), 和 rex_timed_wait()。

Rex_tick（）

Purpose

它周期性地被 ISR 调用。每次调用时，REX 就遍历计时器列表，相应地每个计时器减 1， 并检查哪个计时器期满。

Parameters

Void rex_tick
(
    unsigned long p_ticks
)
函数可以定义无操作。

Rex_def_timer()
Purpose
初始化计时器结构，但不激活计时器。
Parameters
Void rex_def_timer(
        Rex_timer_type   *p_timer,
        Rex_tcb_type   *p_tcb,
        Rex_sigs_type   p_sigs
)

Parameter	How REX uses parameter
P_timer	
P_tcb	P_timer->tcb_ptr=p_tcb
P_sigs	P_timer->sigs=p_sigs

基本算法：
1． 设置 link 和 count 为 0。
2． 设置 TCB, sigs
3． 设在 active 为 FALSE

note：不需要互斥，因为计时器没有激活。INTLOCK 不需要。

Rex_set_timer()
Purpose
激活一个在规定地时间后期满地计时器。注意：注意计时器可能已经在运行，在这种情况下，调用将改变计时器剩余地时间数量。
Parameter
Unsigned long rex_set_timer(
        Rex_timer_type   *p_timer,
        Unsigned long   p_cnt,
)

Parameter	How REX uses parameter
P_timer	要启动的计时器结构。REX 将 cnt 设为 p_cnt 如果计时器不在列表中则将计时器插入列表。
P_cnt	在将计时器插入活动列表之前 REX 装载 p_timer

返回修改前的 cnt 值。老的值或 0。

Rex_get_timer()

Purpose

当允许计时器继续递减时返回计时器的当前计数。

Parameter

Unsigned long rex_get_timer(

Rex_timer_type *p_timer

)

Parameter	How REX uses parameter
P_timer	REX 访问该计时器结构，获取 cnt 值并返回。

Return value	What REX does
Unsigned long	REX 返回 cnt 值

基本算法：

1. Lock interrupts
2. If ther timer is not active, unlock interrupts and return 0.
3. 获取计时器计数，并存储到临时变量中。
4. Unlock interrupts.
5. 返回结果。

Rex_clr_timer()

Purpose

取消一个计时器。计时器停止递减。

Parameters

Unsigned long rex_clr_timer(

Rex_timer_type *p_timer

)

Parameter	How REX uses parameter
P_timer	REX 访问计时器结构，把 cnt 设为 0，然后将其从链接表中移走。

Return value	What REX does
Unsigned long	REX 在零化 cnt 前先行保存，并返回 cnt 的零化前的值。

基本算法：

1. Lock interrupts
2. If timer is not active, unlock interrupts and return 0.
3. 保存 cnt
4. Unlock interrupts
5. Return the milliseconds value.

Rex_timed_wait()

Purpose

可以看作是信号 API 或计时器 API 的部分。

它将挂起一个任务直到给定的信号被设置或者计时器期满。REX 实现是先调用 rex_set_timer(), 然后调用 rex_wait()。

Unsigned long rex_timed_wait(

Rex_sigs_type p_sigs,

Rex_timer_type *p_timer,

Unsigned long p_cnt

)

Parameter	How REX uses parameter
P_sigs	做为 rex_wait()的参数
P_timer	做为 rex_set_timer()的参数。计时器必须已经由 rex_def_timer()初始化完成。
P_cnt	REX 使用其做为 rex_set_timer()的 cnt 参数。

Return value	What REX does
Unsigned long	REX 在零化之前，保存 cnt 并返回原始值。

REX interrupt functions

REX 提供了锁定和解锁中断的能力。提供了两个宏：

- INTLOCK
- INTFREE

- ARM 处理器提供了一个单一的，多用途的中断。
- IRQ 处理器必须查询中断原因以找出发生什么事。

----------这通过读取 MSM 上的中断寄存器完成。

- 中断 trampoline 或 Tramp 读取中断状态并根据它们的软件优先级登记分派的用户安装的中断。
- ISR 处理
  1. IRQ 在硬件里宣称
  2. ARM 切换到 IRQ 模式（保存用户的寄存器）
  3. IRQ 处理器调用 tramp_isr(tramp_init 期间安装)
  4. Tramp_isr 决定哪个中断发生
  5. Tramp_isr 按优先级顺序分派安装的中断
  6. 一旦任何 ISR 被调用，tramp_queue_call 处理一个单一请求的 tramp_queue_call(clock call back)
  7. 控制权返还给 IRQ 处理器。
  8. 控制权返还给就绪的最高优先级的任务。

Common questions

Question 1：两个 REX 任务可以有同样的优先级吗？

No，REX 要求每个任务必须有独立的优先级。否则，调度程序将不能识别。(**将来可能会允许**)

Question 2：如何将自己的任务增加到 REX 启动过程中？

在关于 rex_init()部分，你将看到提供的参数之一就是要启动的第一个任务的 TCB。该任务按次序启动其他的应用程序任务。你可以在该任务中增加代码以启动你希望启动的任务，或者直接在 rex_init()中提供你的新任务的 TCB。 你的新任务还可以启动希望的子任务，然后调用那个 REX 任务，它原先已经在 rex_init()中提供了。.

Question 3: 用户定义的应用程序任务能够有任意的优先级吗？或者优先级必须 REX 任务的优先级低？

优先级的规则如下：两个任务不能有相同的优先级。理论上，用户定义的任务的优先级可以比 REX 高。跟所有的实时系统一样，你必须注意分配优先级以便任务调度同相关的事件的发生一致。比如说吧，很少放弃 CPU 的任务不应当分配比其他所以 REX 任务的优先级都高的优先级--------如果那样做意味着其他 REX 任务在挨饿。

附 1：

任务间通信

- 任务间通信是通过调整队列和信号来完成的。
- 想从其他任务接受消息的任务生成一个全局命令队列和一个与队列相关的信号。
- 想向其它任务发送消息的任务：
  - 获取与目标任务相关的类型的一个命令 buffer.
  - 完成合适区域的填充。
  - 把消息排队到命令队列中。
  - 为该命令队列设置相关的信号。
- 队列操作和信号的设置典型地封装在函数 xxx_cmd()内，这里 xxx 是目标任务的名称。
- 比如：

```
void snd_cmd( snd_packets_type *)
/* accepts packets of type
** snd_packets_type, queues the packet on the snd_cmd_q, and sets the associated command signal
** for the snd task
*/
```

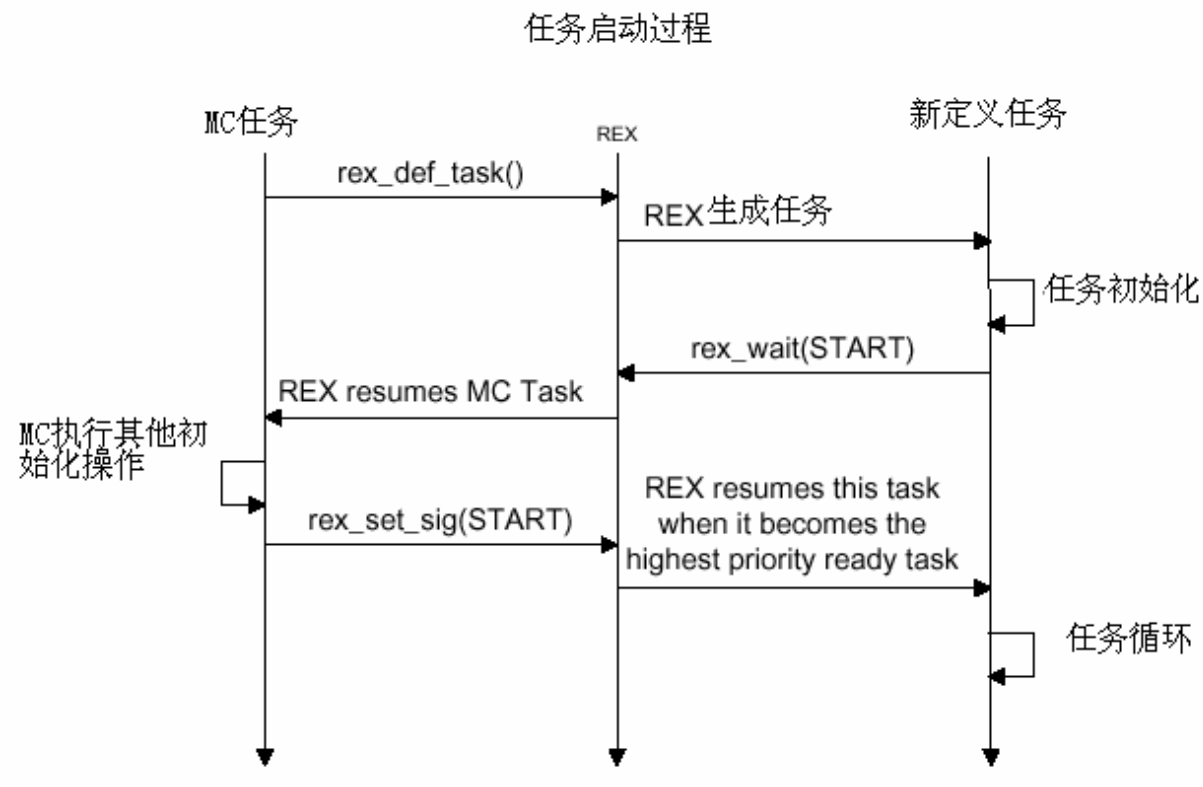
Memory Management via Queues

- 在 DMSS 里，没有专门的存储器管理器。
- 管理存储器的一般方式就是创建一个可使用的存储器模块的队列（一个空闲的模块队列），和已使用的存储器模块(一个正使用的 或命令队列)
- 空闲队列初始化工作如下：

```
my_data_type buffers[10]; q_type my_q;
q_init(&my_q);
for( int x=0;x<10;x++)
    q_put( &my_q, q_link(&buffers[x], &buffers[x]->link ) );
```

任务的启动

- REX 没有对任务的启动做任何限制。
- 一旦由 rex_def_task 启动的任务成为就绪任务中优先级最高的任务，REX 就开始执行该任务。
- 在 REX 中没有暂停后再启动和开始执行命令的概念。
- 在 DMSS 中期望统一各任务的启动以便每个任务都有机会做初始化操作，但它们执行的生命周期由外部控制。
- MC（主控任务）就是负责启动和生命周期的控制。
- 一旦启动，每个任务都可以根据需要做初始化操作。
- 在初始化完成之后，每个任务都必须阻塞并等待来自 MC 的它可以启动的信号。
- 一旦接收到这个信号，任务可以自由开始它的正常处理循环。
- 以下图表给出一个任务的启动过程：



Common services

- queue ----------General purpose FIFO
- bit ----------Bit manipulation routines. Used to pack and unpack bit stream into fundamental type aligned memory
- crc ---------cyclic redundancy check routines, handles CRC16 and CRC30
- ran --------random number generator.
- Qw --------64 –bit(quad word) arithmetic routines
- Misc --------special –purpose register-access routines
- Task --------task control block and stack definitions
- Support headers --------various general definition header files such as:
  >> condef.h -------common global definitions
  >> arm.h -------ARM –specific definitions
  >> processor.h --------general-purpose include to abstract the CPU

Diagnostics

- DIAG 是一个 REX 任务负责处理经串行端口从外部 PC 来的请求。
- DIAG 工作在查询模式，换句话说就是，PC 必须发送请求以从 DMSS 获取数据。
- DIAG 工作使用基于包的协议。

Boot

- Boot 的工作是使你到达函数 main()
- 它也提供了一种检测是否有合法的应用程序驻留在 ROM 中的方法，如果没有，跳到下载器。
- boot 执行以下工作：
  - provides the reset vector
  - sets up exception handlers
  - tests and initializes memory
  - initializes the MSM
  - initializes clocks needed by application –level services to initialize the rest of the MSM

附 2：

具体实现代码

在用户执行时首先就是调用 rex_init(), 如下：

```
int main( void )
{
   rex_init( (void *)irq_stack,          /*  中断栈                  */
             IRQ_Stack_Size,             /*  中断栈大小              */
             &mc_tcb,                    /*  主控任务的 TCB          */
             (void *)mc_stack,           /*  主控任务栈              */
             MC_STACK_SIZ,               /*  主控任务栈的大小        */
             MC_PRI,                     /*  主控任务的优先级        */
             mc_task,                    /*  主控任务入口            */
             0L );                       /*  要传给主控任务的参数*/

   return 0;

} /* end of main */
/*============================================================================
=
```

函数： REX_INIT

描述：初始化 REX. 它初始化内核任务并调用 p_task 主 rex 函数.

依赖关系：

p_tcb 必须是一个有效的任务控制模块(task control block)。

p_task 必须是一个有效的函数指针。.

返回值：无

副作用：

当执行初始化时，该线程将一直处于等待中直到有退出事件被触发，此时它将释放所有资源并返回。

```
============================================================================*
/
void rex_init
(
   void *                  p_istack,          /*  中断栈          */
   rex_stack_size_type     p_istksiz,         /*  中断栈大小      */
   rex_tcb_type           *p_tcb,             /*  任务控制模块    */
   void *                  p_stack,           /*  栈              */
   rex_stack_size_type     p_stksiz,          /*  栈大小          */
   rex_priority_type       p_pri,             /*  任务优先级      */
   void                    (*p_task)( dword ),/*  任务入口函数    */
   dword                   p_param            /*  入口参数        */
)
{
```

```
  /*-------------------------------------------------------
  ** Change to Supervisor mode
  **-------------------------------------------------------*/
#ifndef T_WINNT
  (void) rex_set_cpsr( PSR_Supervisor | PSR_Irq_Mask | PSR_Fiq_Mask);
#endif

  /*-------------------------------------------------------
  **  设置中断栈
  **-------------------------------------------------------*/

  INTLOCK();
  /*  指向栈的顶部  */

  rex_int_stack = (rex_stack_word_type *) p_istack;

  /*-------------------------------------------------------
  ** Initialize the interrupt nest level at zero
  **-------------------------------------------------------*/
  rex_int_nest_level    = 0;

  /*-------------------------------------------------------
  **  初始化计时器列表为空
  **-------------------------------------------------------*/
  rex_null_timer.cnt                    = 0;
  rex_null_timer.tcb_ptr                = NULL;
  rex_null_timer.sigs                   = 0x0;

  rex_null_timer.link.next_ptr          = NULL;
  rex_null_timer.link.prev_ptr          = &rex_timer_list;
  rex_timer_list.link.next_ptr          = &rex_null_timer;
  rex_timer_list.link.prev_ptr          = NULL;

  /*-------------------------------------------------------
  **  初始化任务列表
  **-------------------------------------------------------*/
  rex_task_list.link.next_ptr = &rex_kernel_tcb;
  rex_task_list.link.prev_ptr = NULL;
  rex_kernel_tcb.link.next_ptr = NULL;
  rex_kernel_tcb.link.prev_ptr = &rex_task_list;

  /*-------------------------------------------------------
  ** rex_curr_task  必须跟  rex_best_task 一样以避免此时发生调度
```

```
   **-----------------------------------------------*/
   rex_curr_task = &rex_kernel_tcb;
   rex_best_task = &rex_kernel_tcb;

   rex_def_task(
      &rex_kernel_tcb,                    /* tcb             */
      (void * ) &rex_kernel_stack[0],     /* stack           */
      REX_KERNEL_STACK_SIZE,              /* stack size */
      0,                                  /* priority      */
      rex_idle_task,                      /* function      */
      0                                   /* arguments     */
   );

   /*-----------------------------------------------------
   ** rex_curr_task  必须等于 rex_best_task 以避免此时发生调度
   **-----------------------------------------------*/
   rex_curr_task = p_tcb;
   rex_best_task = p_tcb;

   /*-----------------------------------------------------
   **  定义用户的第一任务为 MC 任务
   **-----------------------------------------------*/
   rex_def_task(
      p_tcb,                              /* tcb             */
      p_stack,                            /* stack           */
      p_stksiz,                           /* stack size */
      p_pri,                              /* priority      */
      p_task,                             /* function      */
      p_param                             /* arguments     */
   );

   INTFREE( );

   rex_start_task( p_tcb );        /*  开始装载任务上下文  */

} /* END rex_init */

;===============================================================================
==
;
; 函数 rex_sched
;
; 描述
```

```
;    该函数执行实际的任务切换. 该函数只能被其它的 rex 函数调用而用户永远不能调用它.
;
;    典型地, 在 REX 服务函数改变了最好任务指针后 rex_sched 被立即调用。
;    比如，当优先级比当前允许任务更改的任务重新激活时，调度发生。
;    Rex_sched 会判断是否当前的任务跟最好的任务指针指的是同一个任务
;    如果那样, 当前任务仍然是最好的任务，不会发生进一步的调度
;   Rex_sched 仅仅返回.
;
;    否则，当前任务不再是最好的任务，Rex_sched 先设置当前的任务跟最好的任务指针等同
;   然后判断它是从 task level 还是从 interrupt level 被调用.
;    如果从 task level 调用, rex_sched 保存老的上下文并装载新的当前任务的上下文。
;    如果从 interrupt level 调用, rex_sched 不好执行上下文切换;直到返回到 task level.
;
; 正式参数：
;    无
;
; 依赖关系：
;    Rex_sched 加到下面用到的全局变量 rex_curr_task, rex_best_task 有合理的值：.
;
; 返回值：
;    无
;
; 副作用：
;    Rex_sched 可能会设置当前的任务指针(rex_curr_task)跟最好的任务指针(rex_best_task)
;   等同如果两者不同的话。
;   而且, rex_sched 可能从老的当前任务到新的当前任务的切换。
;
;================================================================================
==
 LEAF_NODE rex_sched

      mrs     a3, CPSR                      ; Save the CPSR for later.

      orr     a1, a3, #PSR_Irq_Mask    ; locks interrupts
      msr     CPSR_c, a1

;-------------------------------------------------------------------------
; If we are in interrupt level, just return
;-------------------------------------------------------------------------
      and     a1, a3, #PSR_Mode_Mask
      cmp     a1, #PSR_Supervisor       ; If not in Supervisor mode, do not swap
      bne     rex_sched_exit_1          ; until we revert back to task level
```

```
;---------------------------------------------------------------------------
; If a TASKLOCK is in effect, just return
;---------------------------------------------------------------------------
    ldr     a4, =rex_defer_sched        ; deferred sched flag address
    mov     a2, #1                          ; defer sched flag == TRUE
    strb    a2, [a4]                    ; turn flag on

                                            ; test for TASKLOCK
    ldr     a2, =rex_sched_allow        ; load scheduling flag
    ldrb    a2, [a2]                    ; dereference sched. flag
    cmp     a2, #0                          ; compare with FALSE
    beq     rex_sched_exit_1            ; return

    mov     a2, #0                          ; If not TASKLOCK, then fix up the
    strb    a2, [a4]                    ; defer_sched flag.

;---------------------------------------------------------------------------
; Make sure we need to task swap
;---------------------------------------------------------------------------
      ldr     a2, =rex_best_task        ; load the best task into a2
      ldr     a2, [a2]                  ; dereference best task
      ldr     a4, =rex_curr_task        ; load the current task into a4
      ldr     a1, [a4]                  ; dereference current task
      cmp     a2, a1                        ; if current task == best task just return
      beq     rex_sched_exit_1

;---------------------------------------------------------------------------
; Set the curr_task to the new value
;---------------------------------------------------------------------------
      str a2, [a4]                  ; set rex_curr_task=rex_best_task
      mov a4, a1                        ; a4 points now to the last (former current) task

;---------------------------------------------------------------------------
; Increment the slice count.  增加时间片
;---------------------------------------------------------------------------
      ldr     a1, [a2, #REX_TCB_SLICES_OFFSET]      ; load up the slice count
      add     a1, a1, #1                            ; increment it
      str     a1, [a2, #REX_TCB_SLICES_OFFSET]      ; store it

; ---------------------------------------------------------------------
; Save volatile context of CPU  保存 CPU 的上下文
; ---------------------------------------------------------------------
      stmfd     sp!, {lr}                   ; Return address.
```

```
        sub         sp, sp, #8                  ; no need to store r12,r14 in task context.
        stmfd       sp!, {r4-r11}
        sub         sp, sp, #16                 ; Subtract a1-a4 location
#ifdef __APCS_INTERWORK
        tst         lr, #1                      ; Test for thumb return address
        orrne       a3, a3, #PSR_Thumb_Mask ; Return in Thumb mode
#else
        orr         a3, a3, #PSR_Thumb_Mask ; Return in Thumb mode
#endif
        stmfd       sp!, {a3}                   ; First line on rex_sched saves CPSR in a3!!!

;--------------------------------------------------------------------------
; Save the context on stack
;--------------------------------------------------------------------------
        str         sp, [a4,    #REX_TCB_STACK_POINTER_OFFSET]
        mov         a1, a2                      ; a1 = the current task

;--------------------------------------------------------------------------
; rex_start_task_1 is an alternate entry point.
; void rex_start_task(rex_tcb_type *);
; This implies that a1 = current task tcb pointer.
;--------------------------------------------------------------------------
rex_start_task_1

#if defined TIMETEST
; ------------------------------------------------------------------
; TIMETEST - writing current task leds on timetest port.
; ------------------------------------------------------------------
        bl          func_timetest
#endif

; ------------------------------------------------------------------
; Restore the user state, note this may not be the state saved above
; since the call the rex_sched may have changed which stack the handler
; is working on. Note, a context switch will happen here.
; ------------------------------------------------------------------

#if defined(FEATURE_STACK_CHECK)
        ldr         sl, [a1, #REX_TCB_STACK_LIMIT_OFFSET]           ; Stack Limit.
#endif
        ldr         sp, [a1, #REX_TCB_STACK_POINTER_OFFSET]       ; Load the stack pointer
        ldmfd       sp!, {a1}                                       ; Restore SPSR (in a1)
        msr         SPSR_f, a1                                  ; Load SPSR
```

```
        msr         SPSR_c, a1                          ; Load SPSR
        mov         a1, sp                              ; Load sp in a1.
        add         sp, sp, #&3c                        ; adjust {r0-r7, r12, lr, pc}
        ldmfd       a1, {r0-r12,lr,pc}^         ; Load and return, sp already adjusted.

; ------------------------------------------------------------------
; If no context swap occurred, the execution exits thru here
; ------------------------------------------------------------------

rex_sched_exit_1
        msr     CPSR_f, a3                  ; Restore interrupts as prior to rex_sched
        msr     CPSR_c, a3                  ; Restore interrupts as prior to rex_sched

  LEAF_NODE_END

; END rex_sched

;=============================================================================
==
;
;   函数：  rex_start_task
;
;   描述：
;       直接产生任务不用保存任何上下文. 当上下文重新安装时 rex_start_task_1  是
;       rex_sched 的入口点,.  仅被  rex_init 用于初始化用户的第一个任务
;       Alternate_entries  是一个  entry_node  宏，它为调度程序
;       创建  Thumb mode  入口点。
;       用户不能使用这个函数.
;
;   依赖关系：
;       rex_curr_task  必须指向用户的第一个任务的 tcb
;
;   返回值：
;       无
;
; 副作用：
;       用户的第一个任务将被产生。
;
;=============================================================================
==

  ENTRY_NODE Alternate_entries
            CODE16
```

```
        EXPORT    rex_start_task
rex_start_task
        ldr       a4, =rex_start_task_1
        bx         a4

 ENTRY_NODE_END
```

附 3

整个 DMSS 软件流程

main()仅调用 rex_init()，如果 rex_init()完成，整个软件结束。

rex_init()初始化 rex_task_list 和 rex_timer_list 并调用 rex_def_task()定义空闲任务，然后定义 MC 任务，之后调用 rex_start_task()启动主控任务 mc_task()。

1. Mc_task()首先做必要的初始化，比如，自己的各种队列的初始化，中断初始化，时钟初始化等，然后使用 rex_def_task()定义其它的任务，并开始等待被创建任务的反馈 MC_ACK_SIG, 此时 REX 将控制权交给新创建的 task 做初始化，比如创建 sleep 任务，此时 sleep_task()在运行，当 sleep 初始化完成，它首先设置信号 MC_ACK_SIG，告诉 MC 它已经创建完成，并开始等待 TASK_START_SIG，之后 REX 又将控制权交给 MC 任务，依次定义其它的任务。
2. Mc_task()首先清除信号 MC_ACK_SIG，然后按照一定的顺序依次首先启动必要的服务例程。启动就是向各个任务设置 TASK_START_SIG，然后等待 MC_ACK_SIG，当该任务启动完成，即进入了该任务的主循环，它就向 MC 发送 MC_ACK_SIG，然后 MC 接着启动下一个任务。启动顺序是：
   EFS→SFAT→UIM→NV→ERROR SERVICE→USBDC→SLEEP→QDSP→VOC→SND→VOICE SERVICE→TDSO→HS→DIAG→PS→SECURITY SERVICE→CM→UI→PDSM→BT
3. 做必要的参数初始化，即读 NV 数据，做 battery，therm，HW，RF 参数的读取，并根据参数做处理：关机，报错，或者继续启动其它任务。
4. 启动其它的任务：TX→RX→SRCH→RXTX→DS→AUTH→DH→DOG
5. 装载 CDMA 启动必要的网络参数，如果读取失败，进入 offline 状态。
6. 进入模式选择，offline，online，power down 还是 reset。
7. 进入系统决定。