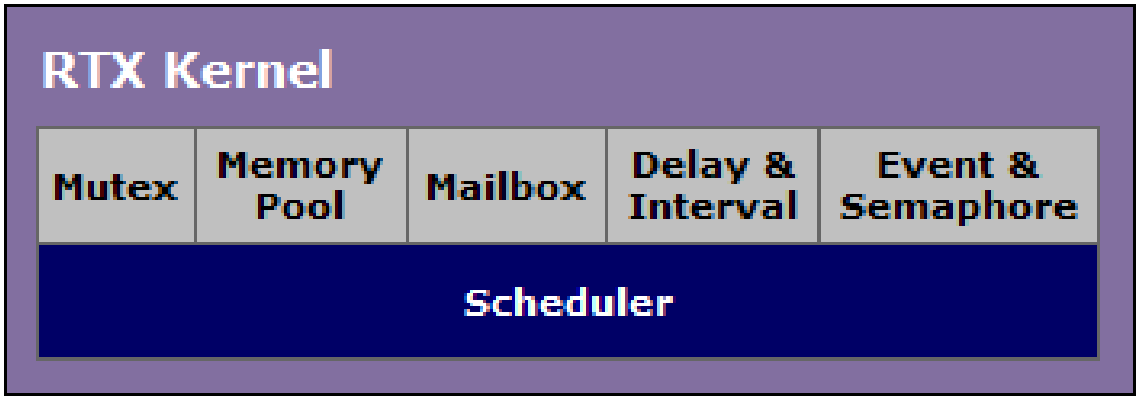
Real-Time 库:
RTX 内核

基于RTX 的内核

- “RTX 内核” (**R**eal-**T**ime e**X**ecutive) 是一个实时操作系统



- “RTX 内核” 支持ARM7、ARM9、Cortex-M3。

Source: http://www.keil.com

基于RTX 内核

- 允许创建多任务的应用程序
- 允许系统资源弹性使用，如CPU存储器
- RTX内核是一个静态系统
- 要想应用程序中使用RTX内核，必须加RTX库（可以使用菜单“options”自动加载）
- 以下两个版本都可以使用
  - ARM7™ / ARM9™
  - Cortex™-M3

基本功能

- 开始实时执行
- 创建任务
- 开始/停止任务执行
- 从一个任务到另一个任务转换
- 实现任务间的通讯

进程间的通信

提供一个任务间通信的机制:

- “事件标志”
- “信号量”
- “互斥”
- “信箱”

RTX Kernel / Technical data

Description	ARM7™/ARM9™	Cortex™-M
Defined Tasks	Unlimited	Unlimited
Active Tasks	250 max	250 max
Mailboxes	Unlimited	Unlimited
Semaphores	Unlimited	Unlimited
Mutexes	Unlimited	Unlimited
Signals / Events	16 per task	16 per task
User Timers	Unlimited	Unlimited
Code Space	<4.5 Kbytes	<4.0 Kbytes
RAM Space for Kernel	350 bytes + 96 bytes Task's stack	300 bytes + 128 bytes Main Stack
RAM Space for a Task	TaskStackSize + 52 bytes	TaskStackSize + 52 bytes
RAM Space for a Mailbox	MaxMessages * 4 + 16 bytes	MaxMessages * 4 + 16 bytes
RAM Space for a Semaphore	8 bytes	8 bytes
RAM Space for a Mutex	12 bytes	12 bytes
RAM Space for a User Timer	8 bytes	8 bytes
Hardware Requirements	One on-chip timer	SysTick timer
User task priorities	1 - 254	1 - 254
Context switch time	<7 µsec @ 60 MHz	<4 µsec @ 72 MHz
Interrupt lockout time	3.1 µsec @ 60 MHz	Not disabled by RTX

Source: http://www.keil.com

RTX Kernel / Time

Function	ARM7™/ARM9™ (µs @60MHz)	Cortex™-M (µs @72MHz)
Initialize system (os_sys_init), start task	46.2	22.1
Create defined task, no task switch	17.0	8.1
Create defined task, switch task	19.1	9.3
Delete task (os_tsk_delete)	9.3	4.8
Task switch (by os_tsk_delete_self)	11.2	5.1
Task switch (by os_tsk_pass)	6.6	3.9
Task switch (upon set event)	7.8	4.3
Task switch (upon sent semaphore)	7.7	4.4
Task switch (upon sent message)	8.4	4.8
Set event (no task switch)	2.4	1.9
Send semaphore (no task switch)	1.7	1.6
Send message (no task switch)	4.5	2.5
Get own task identifier (os_tsk_self)	1.2	1.5
Interrupt response for IRQ ISR	1.1	-
Maximum interrupt lockout for IRQ ISR's	3.1	-
Maximum interrupt latency for IRQ ISR's (response + lockout)	4.2	-

Source: http://www.keil.com

RTX ARM7 & ARM9提示

- IRQ
  - 调度程序可以禁止一小段时间(µs)
- FIQ
  - 不能被禁止
  - 不能从FIQ-ISR调用专用系统内核函数
- SWI (ARM7 & ARM9 设备的软件中断功能)
  - RTX内核保留SWI 0 到 7
  - SWI功能受中断保护（ FIQ除外）
  - 自我创建的SWI功能必须用于RTX内核的处理
- os_clock_demon 任务

RTX Cortex-M3提示

- IRQ
  - 不能由RTX调度程序禁止
- 无FIQ
- SVC (Software Interrupt Function for Cortex-M devices)
  - SVC 0保留给RTX内核
  - 可以中断SVC函数
  - 自我创建的SVC功能必须用于RTX的SVC处理
- 无os_clock_demon任务
- 栈空间
  - 最小128 Byte (256 Byte)

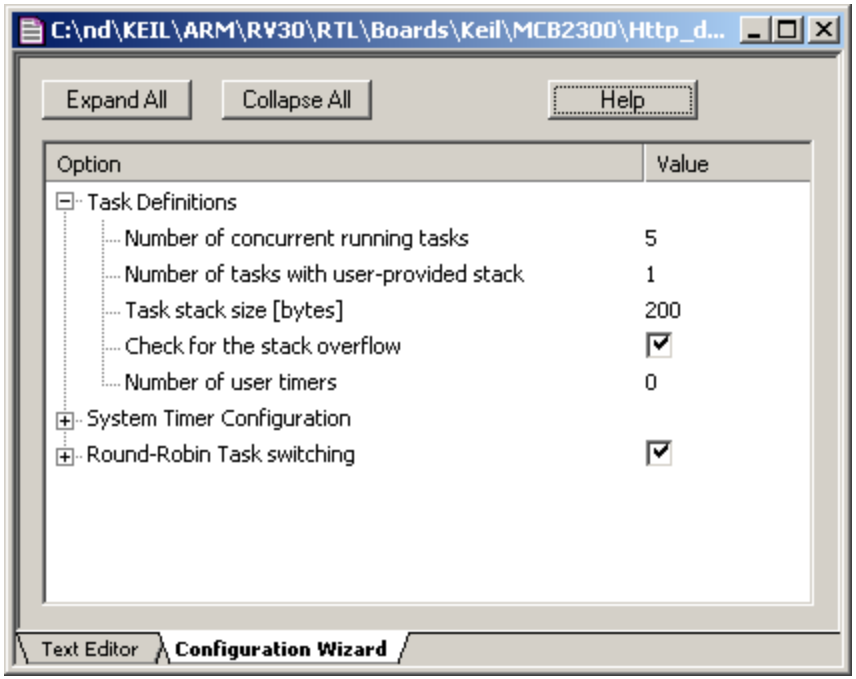
RTX 内核:
任务的使用

Tasks - Basics

- 应用程序的每个作业都会由分割的任务来处理
- 每个外设也可由几个分割的任务来处理
- 任务间进程的通信
- 几个任务可以被“同步”执行
- 每个任务都会有自己的优先权等级
- 任务可以移植到其他应用程序中

系统资源

- 在文件“RTX_Config.c”中进行配置
  - 当前运行的任务数量
  - 用户定义栈的任务数量
  - 默认的栈空间



定时器的时间片中断

- RTX内核需要的周期（按时间片）
- ARM7 和 ARM9 设备上的RTX内核：
  - 使用ARM的其中一个标准定时器
- Cortex-M的RTX内核：
  - 使用系统定时器（ Cortex-M CPU提供专用的RTOS定时器 ）

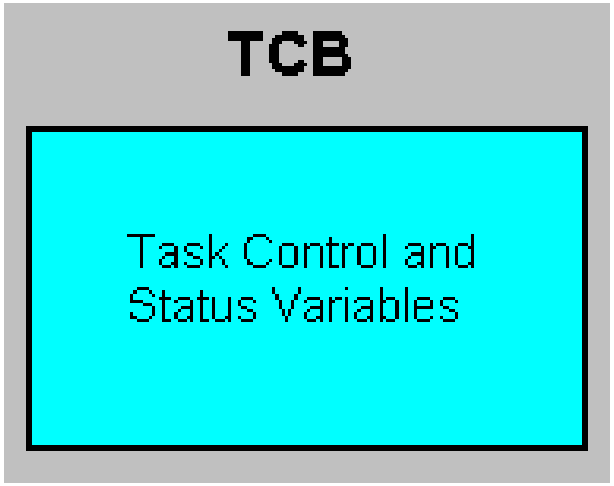
RTX内核默认的任务

- os_clock_demon:
  - 系统时间片定时器任务
  - 调用时间片定时器中断 (ARM7 & ARM9)
  - 管理其他所有任务（延时溢出，唤醒任务，挂起任务等）
  - 具有最高的优先权
  - Cortex-M中不使用RTX库 （Cortex-M 设备有扩展的RTOS特性）
- os_idle_demon:
  - RTX内核的空闲任务
  - 如果没有任务运行RTX内核，转换到
  - 优先权最低

任务控制块

- 每个任务由任务控制块定义**T**ask **C**ontrol **B**lock (TCB)

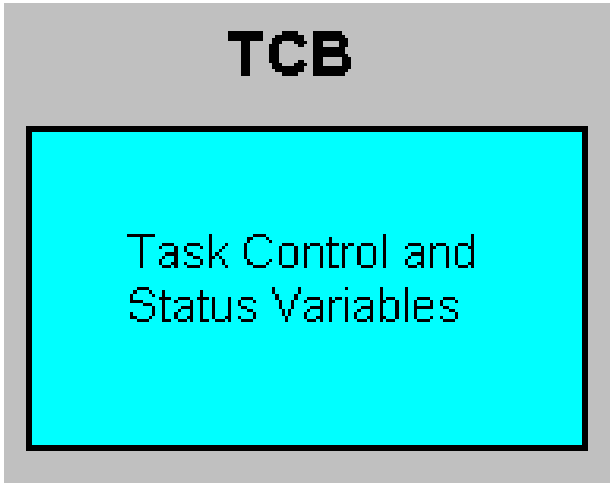
- 在配置文件“RTX_Config.c”中定义TCB存储池的大小
  - 依据并发执行的任务数量来定义



Source: http://www.keil.com

任务控制块

- 任务的TCB在创建任务后，根据任务的运行时间从存储池中动态分配
- 任务的定位信息:
  - 任务控制变量
  - 任务状态变量

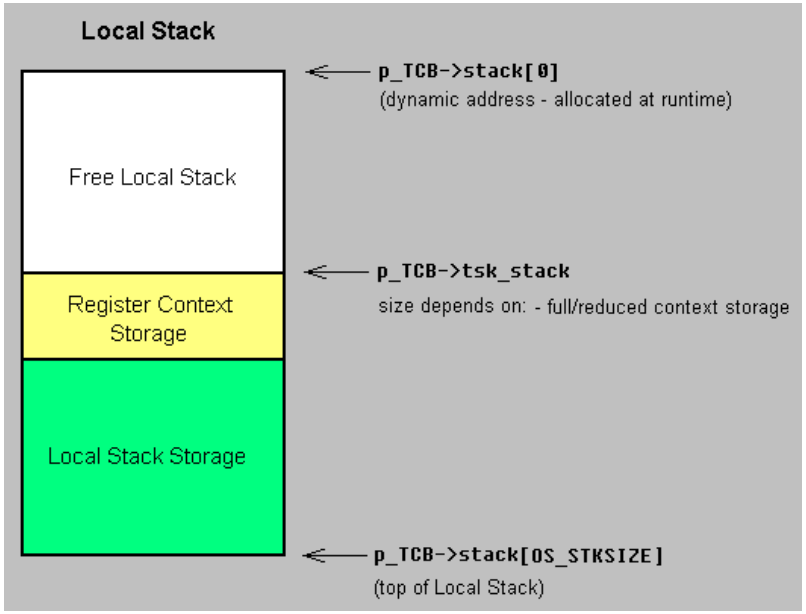


Source: http://www.keil.com

栈管理

- RTX内核为所有任务的栈分配存储池

- 存储池的大小取决于
  - 默认栈的大小
  - 当前运行任务的数量
  - 用户自定义栈的任务数量
  - 设备类型 (Cortex-M or ARM7/9)



Source: http://www.keil.com

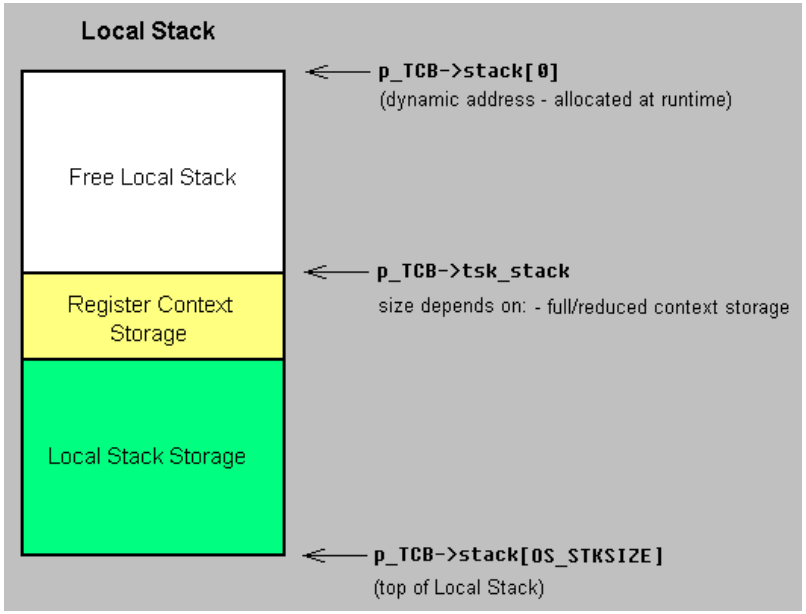
栈管理

- 附加栈:
  - os_clock_demon 任务和
  - os_idle_demon任务
- Cortex-M:

MemoryPoolSize =
(OS_TASKCNT – OS_PRIVCNT + 1) * StackSize

- ARM7 & ARM9:

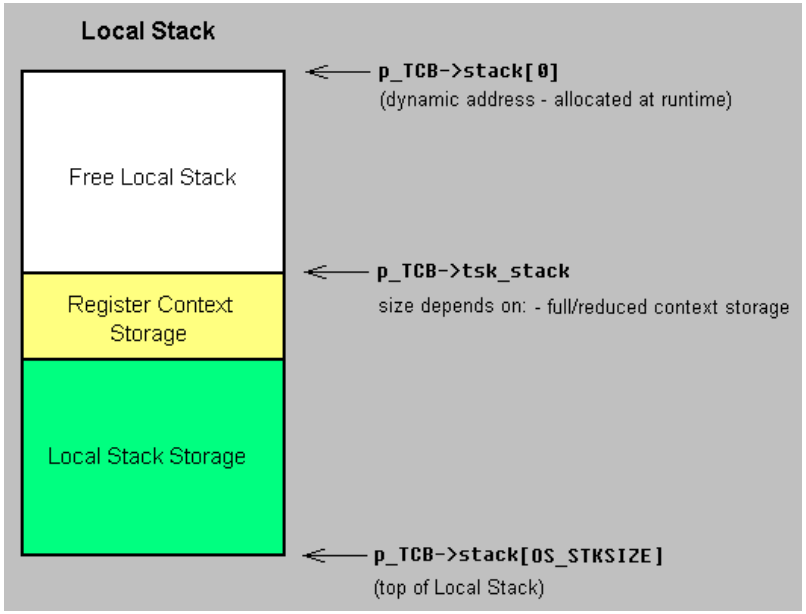
MemoryPoolSize =
(OS_TASKCNT – OS_PRIVCNT + 2) * StackSize



Source: http://www.keil.com

栈管理

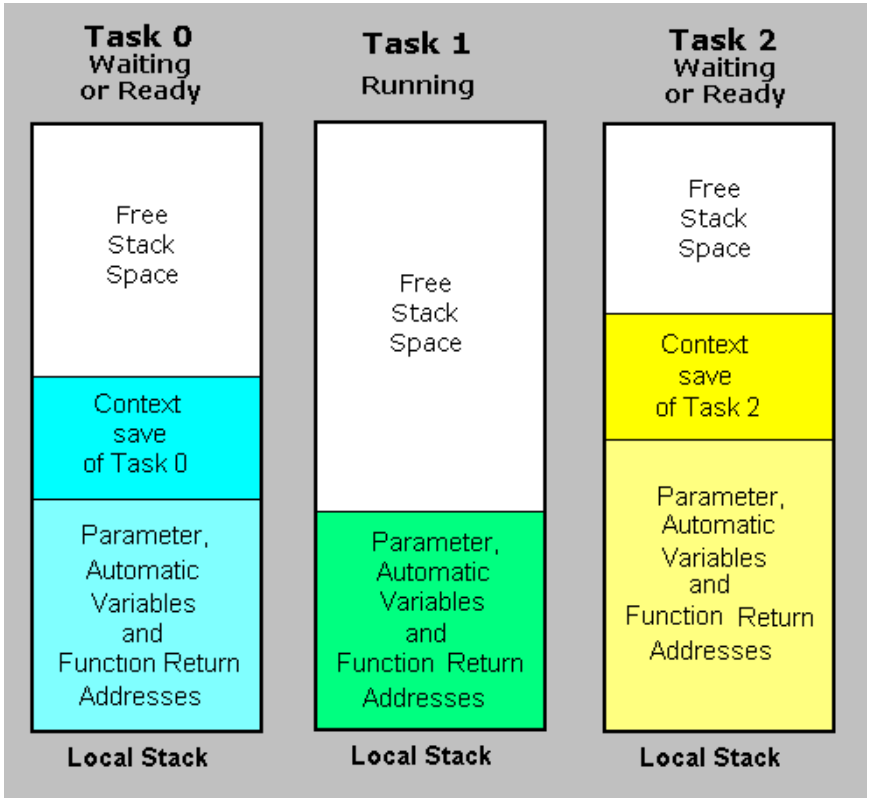
- 任务创建以后，在运行时任务的栈由存储池动态分配
- 用户定义的栈必须由程序分配，必须由新创建的任务指定
- 栈存储块分配以后，指向其位置的指针必须写入到TCB



Source: http://www.keil.com

栈管理

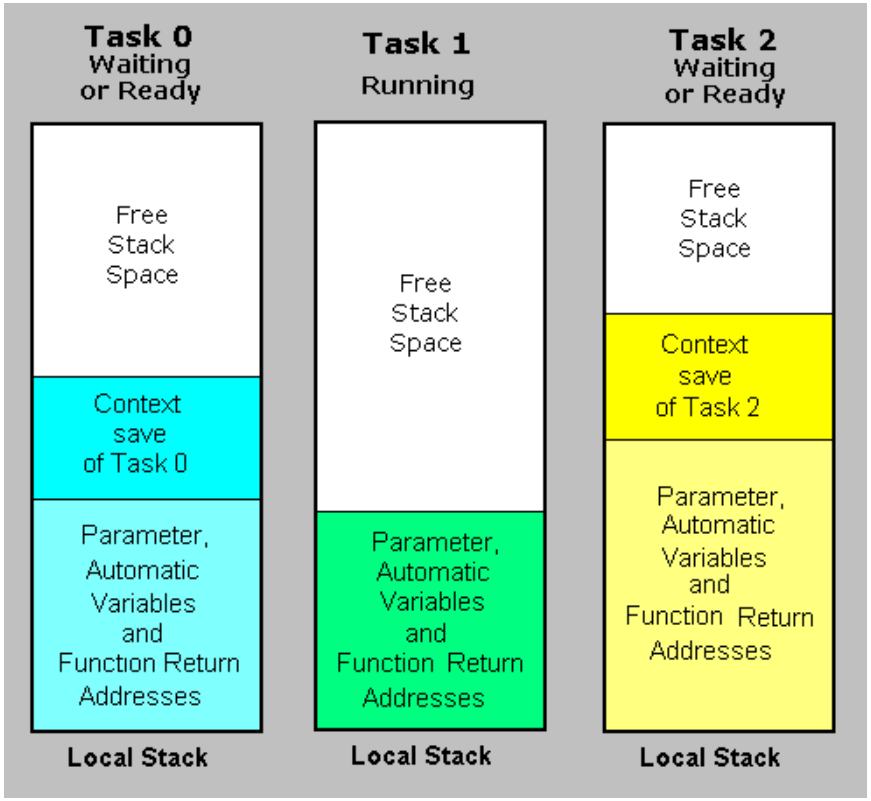
- 每个任务获得自己的栈并进行存储:
  - 参数
  - 变量
  - 函数返回
  - (现场保存)
- 任务转换:
  - 当前运行任务的现场将会保存在本地栈中
  - 切换到下一个任务
  - 保存新任务的现场
  - 新任务开始运行



Source: http://www.keil.com

栈管理

- RTX内核提供了栈溢出校对
  - 在文件RTX_Config.c中允许/禁止此功能
  - 当栈溢出时，RTX核进入函数stack_error_function()中，这个函数是一个死循环



Source: http://www.keil.com

系统启动

ARM7 & ARM9:

- 用户/系统的栈定义在文件**Startup.s中**
- 为RTX内核配置的最小栈空间为:
  - 管理模式　32 bytes　(0x00000020)
  - 中断模式　64 bytes　(0x00000040)
  - 用户模式　80 bytes　(0x00000050)

- 第一个栈创建并启动后方可使用用户模式栈（**User Mode Stack**）

- 用户使用自己创建的中断时，必须要增加中断模式栈的空间（**Interrupt Mode Stack**）
- 用户使用自定义的SWI功能时，必须增加管理模式栈的空间（**Supervisor Mode Stack**）

系统启动

Cortex-M3:

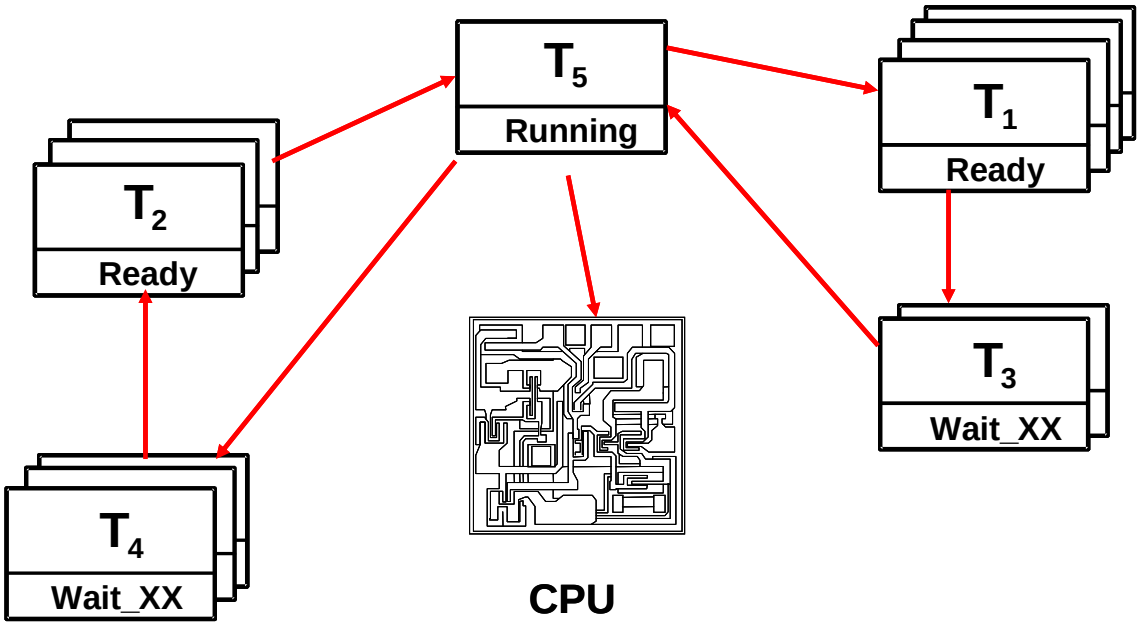
- 在文件**Startup.s**中对主栈的大小进行配置
- RTX核使用系统服务调用（**S**ystem Ser**V**ice **C**alls）
  - 所有SVC函数都使用主栈
  - 使用RTX核时主栈最小为128 bytes
  - 如果使用了中断，主栈的大小建议使用256 bytes
- 如果应用程序使用自有的SVC函数，应再增加主栈的大小

定义

- 多任务是按照操作系统的方式来处理多任务的同步，允许多任务共享系统资源。
- 单CPU系统中，在同一时刻只允许处理一个任务。在各任务之间快速转换，感觉上是多任务在同步执行。
- 几套多任务策略。

任务状态

- 每个任务处于以下几个状态之一:
  - RUNNING
  - READY
  - INACTIVE
  - WAIT_DLY
  - WAIT_ITV
  - WAIT_OR
  - WAIT_AND
  - WAIT_SEM
  - WAIT_MUT
  - WAIT_MBX



任务状态

- RUNNING:
  - 当前运行的任务
  - 同一时刻只有一个任务处于这一状态
  - 当前CPU处理的正是这个任务
- READY:
  - 任务处于准备运行状态
- INACTIVE:
  - 任务还没有被执行或者是任务已经取消

任务状态

- WAIT_DLY:
  - 任务等待延时后再执行
- WAIT_ITV:
  - 任务等待设定的时间间隔到后再执行

任务状态

- WAIT_OR:
  - 任务等待最近的事件标志

- WAIT_AND:
  - 任务等待所有设置事件标志

任务状态

- WAIT_SEM:
  - 任务等待从同步信号发来的“标志”
- WAIT_MUT:
  - 任务等待可用的互斥量
- WAIT_MBX:
  - 任务等待信箱消息或者等待可用的信箱空间来传送消息

多任务策略

- 协同多任务处理
- 抢占多任务处理
- 循环多任务处理

协同多任务处理

- 任务调用自己的调度程序或者使自己进入休眠状态
- 由程序员设计多任务
- 操作系统不影响任务间的转换

Functions

协同多任务处理

- os_tsk_pass()
  - 任务调用调度程序
  - 调度程序决定下一个运行的任务

- os_dly_wait()
  - 任务进入休眠，休眠时间预先设定
  - 调度程序决定下一个运行的任务

抢占多任务处理

- 调度程序可以抢占当前运行的任务
- 调度程序决定下一个运行的任务
- 因此，高优先权任务可以抢占低优先权任务
- 当前的现场要进行保存

循环多任务处理

- CPU时间被分割为时间片。
- 处于“ready”状态的任务（与其他具有相同优先权的任务相比），调度程序会将之优先执行。
- 使用类似os_tsk_pass()的方法，可以将循环多任务处理与协同多任务处理结合起来。
- 当前运行的任务切换到等待状态的时候，由调度程序决定下一个运行的任务。

Example

例 1

- 两个任务有相同的优先权
- 循环模式被禁止用示波器检测 “portpin”
- 会有什么情况发生？

Task 1:

```
void task1 (void) __task {
  while(1) {
    portpin = 1;
  }
}
```

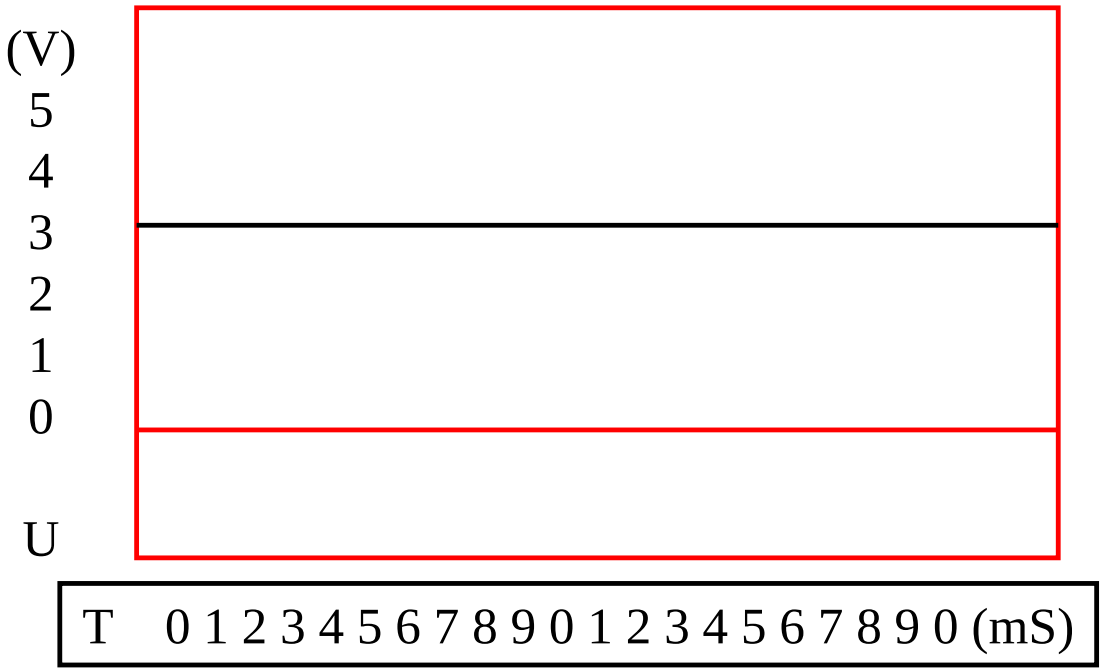
Task 2:

```
void task2 (void) __task {
  while(1) {
    portpin = 0;
  }
}
```

Example

例 1

- 答案: Nothing!
- 第一个任务获得CPU，并一直保持
- 任务切换失败



Example

例 2

- 两个任务具有相同的优先权
- 循环调度程序允许
- 用示波器检测 “portpin”
- 发生什么情况?

Task 1:

```
void task1 (void) __task {
  while(1) {
    portpin = 0;
  }
}
```

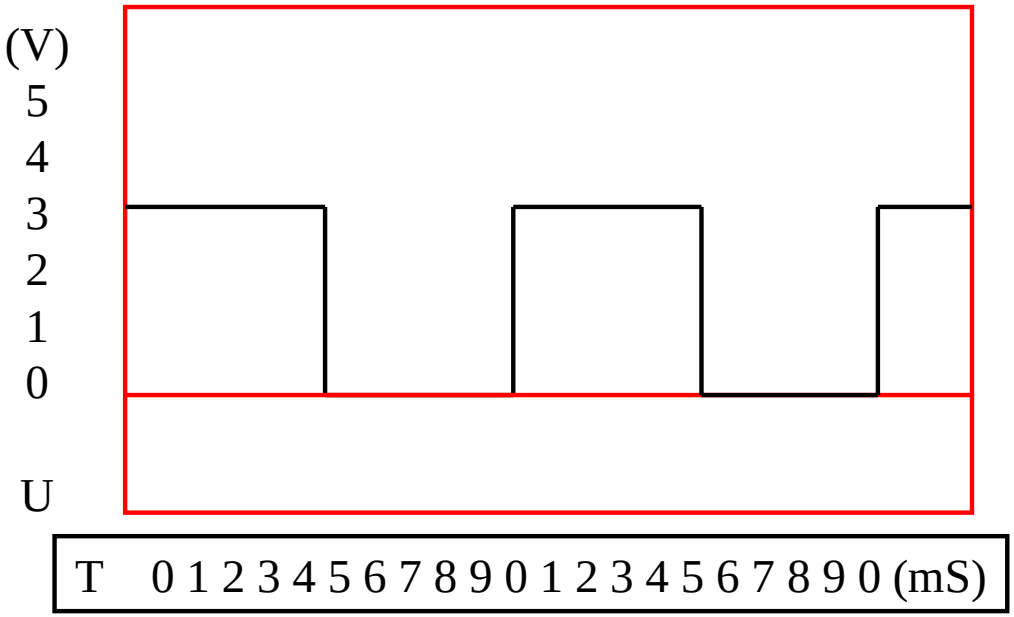
Task 2:

```
void task2 (void) __task {
  while(1) {
    portpin = 1;
  }
}
```

Example

Example 2

- 答案: “portpin” 被触发
- 调度程序每5ms抢占一次

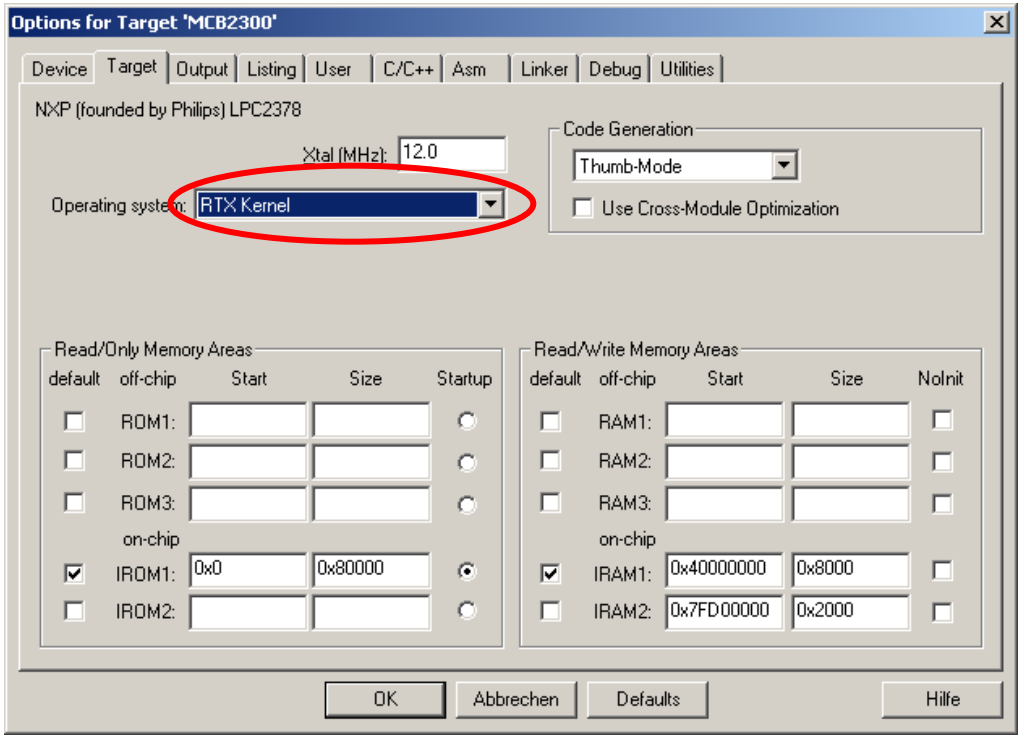


RTX 内核:
配置/ 第一步

Configuration

第一步

- RTX内核的库要链接到工程
- 选择RTX内核作为操作系统，链接就会自动完成

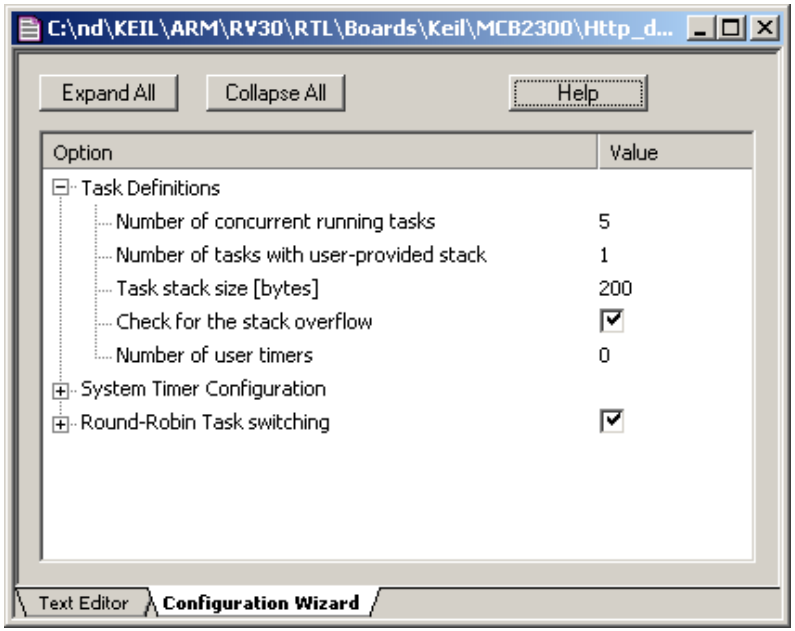


Configuration

第一步

- 将文件RTX_Config.c加入到工程中
  (“\Keil\ARM\Startup”)

- 启动应用程序对应的选项



Configuration

第一步

- 一个简单的任务:

```
void simple_task (void) __task {

   //Initialization

  while(1){

   //Instructions

  }
}
```

- RTX内核从函数os_sys_init(taskname) 开始。

- 这个函数调用定义了由哪个任务开始执行。

RTX 内核:
单元 / 结构方式 / 功能

初始化

初始化并启动 “Real-Time eXecution”

- os_sys_init(Task)
- os_sys_init_prio(Task, Prio)
- os_sys_init_usr(Task, Prio, Stack, Size)
  - 从main()函数调用
  - 系统启动时执行的第一个任务是 “Task”
  - 第一个任务的优先权可以由“Prio”来定义
  - 用户定义的 “栈” 以及对其分配的空间可以指派给第一个任务
- 优先权: 1-254 (最高: 254; 最低: 1)

中断

- 中断服务程序中的特殊函数调用
  - 使用 “isr_” 代替 “os_”

- ARM7 & ARM9:
  - 运行RTX内核时避免使用潜入中断
    (最好使用短 ISR, 并设置为高优先权，在任务中执行指令)
  - 允许FIQs 功能
  - 不要从FIQ-ISRs中调用“isr_”-函数

中断

允许/禁止中断功能

- tsk_lock()
  - 禁止RTX内核定时器中断
- tsk_unlock()
  - 允许RTX内核定时器中断

- 注意：禁止RTX内核定时器中断会阻断调度程序，时间溢出也不起作用

创建

创建一个任务

- os_tsk_create (TaskPtr, Prio)
  - 从TCB存储池分配一个TCB
  - TCB中保存着任务的状态变量和任务的控制变量
  - 任务的初始状态为“ready”
- os_tsk_create_ex (TaskPtr, Prio, *argv)
  - 用这个函数的附加参数创建一个任务

创建

使用用户定义的栈创建任务：

- os_tsk_create_user (TaskPtr, Prio, Stack, Size)
  - 分配TCB并填充其超时
  - 任务的初始状态为“ready”
  - 获得优先权 “Prio”
  - 任务在用户给定的栈空间中运行
- os_tsk_create_user_ex (TaskPtr, Prio, Stack, Size, *argv)
  - 使用这个函数按照用户给定的栈和附加参数创建新任务

删除

删除任务:

- os_tsk_delete(TaskId)
  - 标有“TaskId”的任务将被删除
- os_tsk_delete_self()
  - 当前运行的任务将被删除

优先权

改变任务的优先权:

- os_tsk_prio(TaskId, Prio)
  - 使用这个函数改变标有“TaskId”任务的优先权
  - 这个函数造成了“重调度”（除了协同调度程序以外）
- os_tsk_prio_self(Prio)
  - 使用这个函数来改变当前运行任务的优先权

事件函数

- 每个任务有16个事件标志
- 任务可以等待事件出现
- 任务可以等待一个事件以上的相互结合 (OR/ AND)

等待

等待事件:

- os_evt_wait_or(Mask, Time)
  - 任务等待至少一个事件的发生 (定义为 “Mask”)
  - “Time” 指定为时间溢出，等待事件的发生 (0xffff = 不限制, 0-0xfffe等待时间)

- os_evt_wait_and(Mask, Time)
  - 任务等待所有定义为“Mask”的事件
  - “Time” 指定为时间溢出，等待事件的发生 (0xffff = 不限制, 0-0xfffe等待时间)

获取

Received event flags on os_evt_wait_or:

- os_evt_get()
  - 在函数os_evt_wait_or(Mask, Time)中接收定义为“Mask”的事件标志，并返回其值。

设置

为指定的任务设置事件的flag:

- os_evt_set(Mask, TaskId)
  - 用 “Mask”指定要设置的事件
  - 用 “TaskId”定义任务

从IRQ设置

在ISR中的专用函数来设定事件：

- isr_evt_set(Mask, TaskId)
  - ISR中使用这个函数设定事件（指定为“Mask”)
  - “TaskId” 用来定义目标任务

清除

清除事件的flag：

- os_evt_clr(Mask, TaskId)
  - 用 “Mask”指定要清除的事件
  - 用 “TaskId”定义目标任务

功能

- 不同类型的定时器。
  - 单个短定时器（用户定时器）。
  - 周期定时器。
- 用户定时器可以创建，取消，挂起，重启。
- 用户定时器的在设定的时间到达以后，会调用用户提供的返回函数 os_tmr_call()，完成后将之删除。

用户定时器的创建

创建一个用户定时器:

- os_tmr_create (Time, Info)
  - “Time” 定义了系统时间片溢出的时间
  - 参数 “Info” 定义了用户定时器（将参数传给函数os_tmr_call()）
  - 函数返回定时器的ID

用户定时器的取消

取消用户定时器

- os_tmr_kill (TimerId)
  - 在定时器所设定的时间到来之前，可以将定时器取消
  - “TimerId”定义了要取消的定时器

用户定时器回叫

用户定时器回叫功能:

- os_tmr_call (info)
  - 如果用户定时器设定的时间到时，就会调用这个函数。
  - os_tmr_call(info) 位于文件 RTX_Config.c 中，可以填充用户提供的代码。
  - 参数 “info” 用来识别回叫功能中的定时器。

定时器设置

设置循环定时器:

- os_itv_set (Time)
  - 这个函数设置了定时器的时间片个数 (“Time”)
  - 这个函数不能启动定时器!

定时器等待

启动循环定时器:

- os_itv_wait ()
  - 此函数启动了预先定义的循环定时器 (os_itv_set(Time))
  - 任务等待时间周期的到来

信箱

- 发送消息到信箱，而不是任务
- 一个任务可以有一个以上的信箱
- 消息通过地址传送而不是值传送

创建

创建信箱:

- os_mbx_declare (Mbox, Cnt)
  - 这个宏定义了一个目的信箱（静态矩阵）
  - “Mbox”是信箱的标识符
  - “Cnt”指定了此信箱的消息数量

初始化

初始化信箱:

- os_mbx_init (Mbox, Size)
  - 初始化目标信箱的“Mbox”（在函数 os_mbx_init()中声明）
  - “Size” 指定了信箱的大小

发送

发送消息到信箱:

- os_mbx_send(Mbox, Msg, Time)
  - 如果信箱未满，发送消息 (“Msg”)到信箱 (“Mbox”)
  - “Time” 指定了一个任务等待的信箱时间（按时间片）
  - (0xffff = 未限制, 0-0xfffe 按时间片等待的时间)

从 ISR发送

从ISR发送消息到信箱:

- isr_mbx_send(Mbox, Msg)
  - 使用此函数从ISR发送消息 (“Msg”) 到信箱 (“Mbox”)
  - 注意：这个函数没有时间溢出。
  - 如果信箱中的消息已满，会被内核忽略。
  - 使用函数 isr_mbx_check 来检测信箱是否已满。

接收

在ISR中接收消息:

- isr_mbx_receive (Mbox, Msg)
  - 从信箱 (“Mbox”)接收消息(“Msg”)。
  - 如果消息从信箱中读出，返回 “OS_R_MBX”。
  - 如果没有消息，返回 “OS_R_OK” 。
  - 注意: “Msg” 是 ** !!

等待

从信箱接收消息:

- os_mbx_wait (Mbox, Msg, Time)
  - 用此函数从信箱中接收消息
  - “Time”指定任务从信箱中获取的最大等待时间 （如果信箱为空，任务进入睡眠状态）
  - (0xffff = 未限制, 0-0xfffe 等待时间（以时间片计）, 0 立即继续等待)
  - 注意: “Msg” 是 ** !!

校验

校验信箱的空间:

- os_mbx_check (Mbox)
  - 返回信箱仍可存放的消息个数。
- isr_mbx_check (Mbox)
  - 校验信箱中空余的入口。

互斥

- 缺乏“共同执行的目标”
- 允许多任务共享资源
- 用互斥保护这个“临界区”

初始化

初始化互斥量:

- os_mut_init(Mutex)
  - 初始化指定的互斥 “Mutex” 目标

等待

等待进入请求的资源：

- os_mut_wait(Mutex, Time)
  - 尝试获取互斥信号量 “Mutex” (尝试进入临界区)。
  - 如果互斥信号量表示上锁，任务进入睡眠模式，一直等等信号量解锁再进入临界区 。
  - “Time”指定了任务等待互斥信号量解锁的等待时间
  - (0xffff = 不受限制, 0-0xfffe 等待时间（按时间片）, 0 立即进行)

释放

释放互斥量:

- os_mut_release(Mutex)
  - 使用此函数释放信号量 “Mutex”。
  - 只有任务自己才能释放互斥信号量。

信号量

- 信号量很少在嵌入式系统中使用。
- 信号量经常不正确使用。
- “死锁”和“饿死”的危险。
- 互斥是一个信号量，值为1。

初始化

初始化信号量:

- os_sem_init(Sem, Count)
  - 创建信号量 “Sem”。
  - “Count” 指定信号量的值。

请求

请求信号量：

- os_sem_wait(Sem, Time)
  - 从信号量(“Sem”)获得一个标记。
  - 如果标记不为0，任务继续执行。
  - “Time” 指定了任务等待释放信号量的等待时间。
  - (0xffff = 不受限制, 0-0xfffe 等待时间（按照时间片）, 0 立即继续)

释放

释放信号量:

- os_sem_send(Sem)
- isr_sem_send(Sem)
  - 从ISR给信号量(“Sem”)释放一个标记。

存储管理

- 允许“动态”分配、释放存储空间，尤其是优化嵌入式系统。
- 基于静态矩阵(存储池)的方法。
- 可以为信箱通信分配消息。

申请

申请一个存储池：

- _declare_box(Pool, Size, Count)
  - 此宏用来申请数组。
  - “Size” 指定每块有多少bytes。
  - “Count” 指定块数。
  - “Pool” 指定存储池的名称。
  - 此外，此宏申请了一个12bytes的块，用来存储内部指针和空间大小等信息。
- _declare_box8(Pool, Size, Count)
  - 此宏是8-byte 的对齐方式。

初始化

初始化存储池:

- _init_box(Pool, PoolSize, BlkSize)
  - 以大小为 “PoolSize” 来初始化存储池 “Pool” (bytes)。
  - “BlkSize” 指定了存储池中每块的大小。
- _init_box8(Pool, PoolSize, BlkSize)
  - 此函数用做8-byte对齐。

分配

分配存储池中的一个块：

- _alloc_box(Pool)
  - 此函数用来分配存储池中的一个块。
  - 返回新分配块的指针。
  - 如果没有可以使用的块，返回“NULL”。
  - 这个函数可以再进入，保障线程的安全。
- _calloc_box(Pool)
  - 为“Pool”分配一个块，并初始化为0。
  - 这个函数可以再进入，保障线程的安全。

释放

释放存储池的一个块：

- _free_box(Pool, Block)
  - 释放存储池中不用的块。
  - 这个函数可以再进入，保障线程的安全。

RTX 内核提示

- 用任务构建软件任务
  - 任务少的test比任务多的test要简单的多。
- 不应结束任务

  while (1)
- 每个任务有一个入口
  - 等待事件
  - 等待消息

```
void simple_task (void) __task {

   //Initialisation

  while(1){

   //Wait for event/message

      //Instruction 1
      //Instruction 2
      //  …
      //Instruction N

   //Send event/message

  }
}
```

RTX 内核提示

ARM7 & ARM9:

- 在函数os_sys_init()之前，可以调用简单 IOs的初始化程序。
  - 小函数。
  - 不要使用太多本地变量。
  - 用户模式下的栈可以发生溢出。
- 在第一个任务执行之前，应完成复杂的初始化。

RTX 内核提示

Cortex-M3:

- 在os_sys_init()之前初始化简单的IO。
  - 以特权模式执行这个初始化程序。
  - 推荐在此函数中配置外设。
- 以非特权模式执行任务。

RTX 内核:
调试

退出任务

- 显示所以退出的任务和他们的状态。

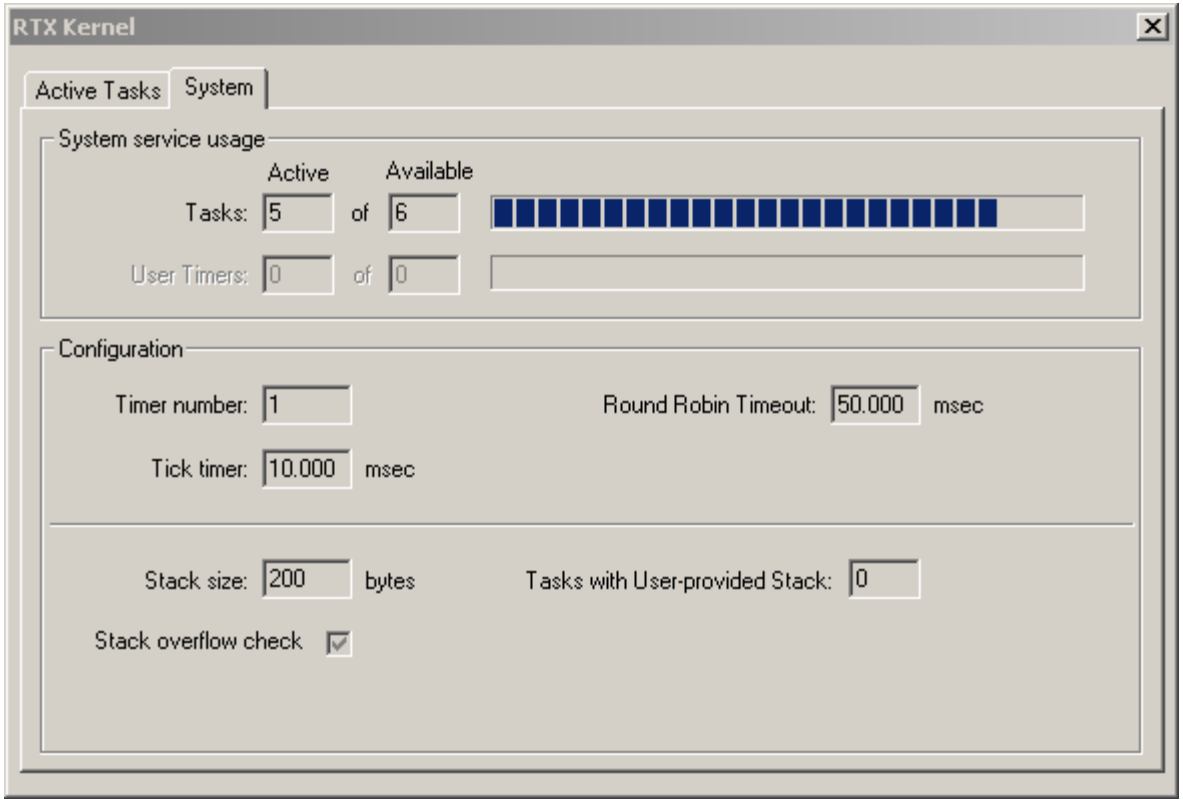
RTX Kernel

Active Tasks | System

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
0	os_clock_demon	255	WAIT_ITV	1			36%
2	phaseA	1	WAIT_DLY	22			52%
3	phaseB	1	WAIT_AND		0x0000	0x0001	48%
4	phaseC	1	WAIT_AND		0x0000	0x0001	48%
5	phaseD	1	WAIT_AND		0x0000	0x0001	48%
6	clock	1	WAIT_AND		0x0000	0x0100	48%
255	os_idle_demon	0	RUNNING				0%

退出任务

- RTX内核的任务数与系统信息。



Next Topic:

RTX Examples for Cortex-M