FreeRTOS(读作"free-arr-toss")是一个嵌入式系统使用的开源实时操作系统。FreeRTOS被设计为“小巧，简单，和易用”，能支持许多不同硬件架构以及交叉编译器。

FreeRTOS自2002年Richard Barry开始开发以来，一直都在积极开发中。至于我，我不是FreeRTOS的开发人员或贡献者，我只不过是一个最终用户和爱好者。因此，这章将着重与FreeRTOS架构之“是什么”和“怎么做”，而相对本书其他章节来说，较少去讲“为什么”。

就像所有操作系统一样，FreeRTOS的主要工作是执行任务。大部分FreeRTOS的代码都涉及优先权、调度以及执行用户自定义任务。但又与所有其他操作系统不同，FreeRTOS是一款运行在嵌入式系统上的实时操作系统。

到本章结束，我希望你可以了解FreeRTOS的基本架构。大部分FreeRTOS致力于执行任务，所以你可以很好地看到它究竟是如何做到的。

如果这是你首次去深入了解一个**操作系统**，我还是希望你可以学习到最基本的操作系统是如何工作的。FreeRTOS是相对简单的，特别是相比Windows，**Linux**，或者OS X而言，不过所有操作系统都有着相同的概念和目标，所以不论学习哪个操作系统都是有启发和有趣的。

3.1 什么是“嵌入式”和“实时”?

“嵌入式”和“实时”对于不同的人来说代表不同的理解，所以让我们像FreeRTOS用户那样来定义它们。

嵌入式系统就是一个专门设计用来做一些简单事情的计算机系统，就像是电视遥控器，车载GPS，电子手表，或者起搏器这类。嵌入式系统比通用计算机系统显著的区别在于更小和更慢，通常也更便宜。一个典型的低端嵌入式系统可能有一个运行速度为25MHz的8位CPU，几KB的内存，和也许32KB的闪存。一个高端的嵌入式系统可能有一个运行速度为750MHz的32位CPU，一个GB左右的内存，和几个GB的闪存。

实时系统是设计去完成一定时间内的事，它们保证这些事是在应该做的时候去做。

心脏起搏器是实时嵌入式系统的一个极好例子。起搏器必须在正确的时间收缩心肌，以挽救你的生命；它不能够太忙而没有及时响应。心脏起搏器以及其他的实时嵌入式系统都必须精心设计，以便在任何时刻都能及时执行它们的任务。

3.2 架构概述

FreeRTOS是一个相对较小的应用程序。最小化的FreeRTOS内核仅包括3个（.c）文件和少数头文件，总共不到9000行代码，还包括了注释和空行。一个典型的编译后（二进制）代码映像小于10KB。

FreeRTOS的代码可以分解为三个主要区块：任务，通讯，和硬件接口。

●任务：大约有一半的FreeRTOS的核心代码用来处理多数操作系统首要关注的问题：任务。任务是给定优先级的用户定义的C函数。task.c和task.h完成了所有有关创建，调度，和维护任务的繁重工作。

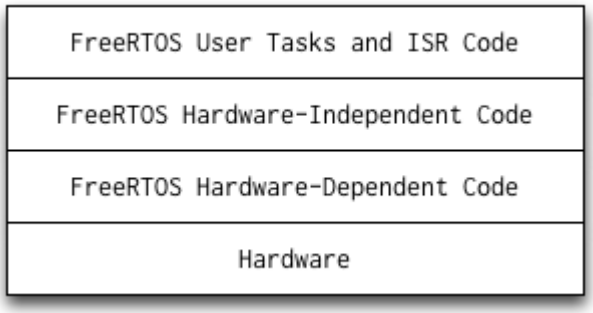
●通讯：任务很重要，不过任务间可以互相通讯则更为重要！它给我们带来FreeRTOS的第二项任务：通讯。大约40%的FreeRTOS核心代码是用来处理通讯的。queue.c和queue.h是负责处理FreeRTOS的通讯的。任务和中断使用队列互相发送数据，并且使用信号灯和互斥来发送临界资源的使用情况。

●硬件接口：接近9000行的代码拼凑起基本的FreeRTOS，是硬件无关的；相同的代码都能够运行，不论FreeRTOS是运行在不起眼的8051，还是最新、最炫的ARM内核上。大约有6%的FreeRTOS的核心代码，在硬件无关的FreeRTOS内核与硬件相关的代码间扮演着垫片的角色。我们将在下个部分讨论硬件相关的代码。

硬件注意事项

硬件无关的FreeRTOS层在硬件相关层之上。硬件相关层声明了你选择什么样的芯片架构。

图3.1显示了FreeRTOS的各层。



FreeRTOS包含所有你需要用来启动很运行系统的硬件无关以及硬件相关的代码。它支持许多编译器（CodeWarrior，GCC，IAR等）也支持许多处理器架构（ARM7，ARM Cortex-M3，PICs各系列，Silicon Labs 8051, x86等）。请参阅FreeRTOS网站，可以看到处理器和编译器支持列表。

FreeRTOS是高可配置设计。FreeRTOS可以被编译成为适合单CPU，极简RTOS，只之支持少数任务的操作系统，也可以被编译成为适合多核功能强大的结合了TCP/IP，文件系统，和USB的怪兽。

配置选项可以通过设置不同的#defines，在FreeRTOSConfig.h文件里选择。时钟速度，堆大小，互斥，和API子集，连同其他许多选项，都可以在这个文件中配置。这里是几个例子，设置了任务优先级的最大数量，CPU的频率，系统节拍器的频率，最小的堆栈大小和总的堆大小：

```cpp
#define configMAX_PRIORITIES      ( ( unsigned portBASE_TYPE ) 5 )
#define configCPU_CLOCK_HZ        ( 12000000UL )
#define configTICK_RATE_HZ        ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE  ( ( unsigned short ) 100 )
#define configTOTAL_HEAP_SIZE     ( ( size_t ) ( 4 * 1024 ) )
```

对于不同的交叉编译器和CPU架构，硬件相关代码分布在多个文件中。举例来说，如果你使用ARM Cortex-M3芯片，IAR编译器工作，那么硬件相关的代码就存在FreeRTOS/Source/portable/IAR/ARM_CM3/目录下。portmacro.h文件声明了所有硬件特定功能，port.c和portasm.s 包含了所有实际硬件相关的代码。硬件无关的头文件portable.h在编译的时候用#include's引入正确的portmacro.h文件。FreeRTOS使用#define'd调用在portmacro.h中声明的硬件特定功能。

让我们来看一个例子，FreeRTOS是如何调用一个硬件相关功能的。硬件无关的文件tasks.c常常需要插入一个代码的临界区，以防止抢占。在不同架构上，插入一个临界区的表现也不同，并且硬件无关的task.c不需要了解硬件相关的细节。所以task.c调用全局宏指令portENTER_CRITICAL(), 乐得忽略它实际上是如何做到的。假设我们使用IAR编译器在ARM Crotex-M3芯片上编译生成FreeRTOS，使用那个定义了portENTER_CRITICAL()的文件/Source/portable/IAR/ARM_CM3/portmacro.h，如下所示：

```cpp
#define portENTER_CRITICAL()   vPortEnterCritical()
```

vPortEnterCritical()实际上是在FreeRTOS/Source/portable/IAR/ARM_CM3/port.c中定义的。这个port.c文件是一个硬件相关的文件，同时包含了对IAR编译器和Cortex-M3芯片认识的代码文件。vPortEnterCritical()函数利用这些硬件特定的知识进入临界区，又返回到与硬件无关的task.c中。

protmacro.h文件也定义了一个数据类型的基本架构。这个数据类型中包括了基本整型变量，指针，以及系统时钟节拍器的数据类型，在ARM Cortex-M3 chips上使用IAR编译器时，就采用如下定义：

```cpp
#define portBASE_TYPE  long              // Basic integer variable type
#define portSTACK_TYPE unsigned long     // Pointers to memory locations
typedef unsigned portLONG portTickType;  // The system timer tick type
```

这样使用数据类型的方法，和函数透过小层的#defines，看上去略微有点复杂，不过它允许了FreeRTOS能够被重新编译在一个完全不同的系统架构上，仅仅只需要通过修改这些硬件相关的文件。同时，如果你想要让FreeRTOS运行在一个当前尚未被支持的架构上，你只仅仅需要去实现硬件相关的功能，这要比在FreeRTOS上去实现硬件无关的部分，要少得多。

就如同我们已经见到的，FreeRTOS用C的预处理宏#define来实现硬件相关的功能。FreeRTOS也同样用#define来应对大量的硬件无关的代码。对于非嵌入式应用程序这样频繁使用#define是一个严重的错误，不过在许多小型嵌入式系统中这点开销比起“实时”所提供的功能来说就微不足道了。

3.3. 调度任务：快速概述

任务优先级和就绪列表

所有任务都有一个用户指定优先级，从0（最低优先级）到 configMAX_PRIORITIES-1的编译时间值（最高优先级）。例如，如果configMAX_PRIORITIES设置为5，当FreeRTOS使用5个优先等级时：0(最低优先级)，1，2，3，和4（最高优先级）。

FreeRTOS使用一个“就绪列表”来跟踪所有已经准备好运行的任务。它像一个任务列表数组来实现就绪列表，如下所示：

```cpp
static xList pxReadyTasksLists[ configMAX_PRIORITIES ];  /* Prioritised ready tasks.  */
```

pxReadyTasksLists[0]是所有准备好的优先级为0的任务列表，pxReadyTasksLists[1]是所有准备好的优先级为1的任务列表，以此类推，直到pxReadyTasksLists[configMAX_PRIORITIES-1]。

系统节拍器（时钟）

FreeRTOS系统的心跳就是被称为系统节拍器（时钟）。FreeRTOS配置这个系统生成一个定期的节拍（时钟）中断。用户可以配置的节拍中断频率，通常是在毫秒范围。vTaskSwitchContext()函数在每次的节拍中断释放的时候被调用。vTaskSwitchContext()选择优先级最高的就绪任务并将它赋予pxCurrentTCB变量，如下所示：

```cpp
/* Find the highest-priority queue that contains ready tasks. */
while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopReadyPriority ] ) ) )
{
    configASSERT( uxTopReadyPriority );
    --uxTopReadyPriority;
}
/* listGET_OWNER_OF_NEXT_ENTRY walks through the list, so the tasks of the same
priority get an equal share of the processor time. */
```

listGET_OWNER_OF_NEXT_ENTRY(pxCurrentTCB, &(pxReadyTasksLists[uxTopReadyPriority]));

在当型循环（while loop）开始之前，uxTopReadyPriority就被确保大于或等于优先级最高的就绪任务。while()循环从优先级uxTopReadyPriority开始，循环走下去从pxReadyTasksLists[]数组里找到就绪任务优先级最高的那个。接着listGET_OWNER_OF_NEXT_ENTRY()就抢占那个就绪列表中优先级最高的下一个就绪任务。

现在pxCurrentTCB指向了优先级最高的任务，并且当vTaskSwitchContext()返回硬件相关代码时开始运行那个任务。

那九行代码是FreeRTOS的绝对核心。其余FreeRTOS的8900+行代码都是用来确保那九行代码，全都是用来保持优先级最高任务的运行的。

图3.2是一个大致的就绪列表看起来像什么的图。这个例子有三个优先级，有一个优先级为0的任务，没有优先级为1的任务，和三个优先级为2的任务。这张图是准确的，但不完整的;它的少掉一些细节，我们稍后将补充。

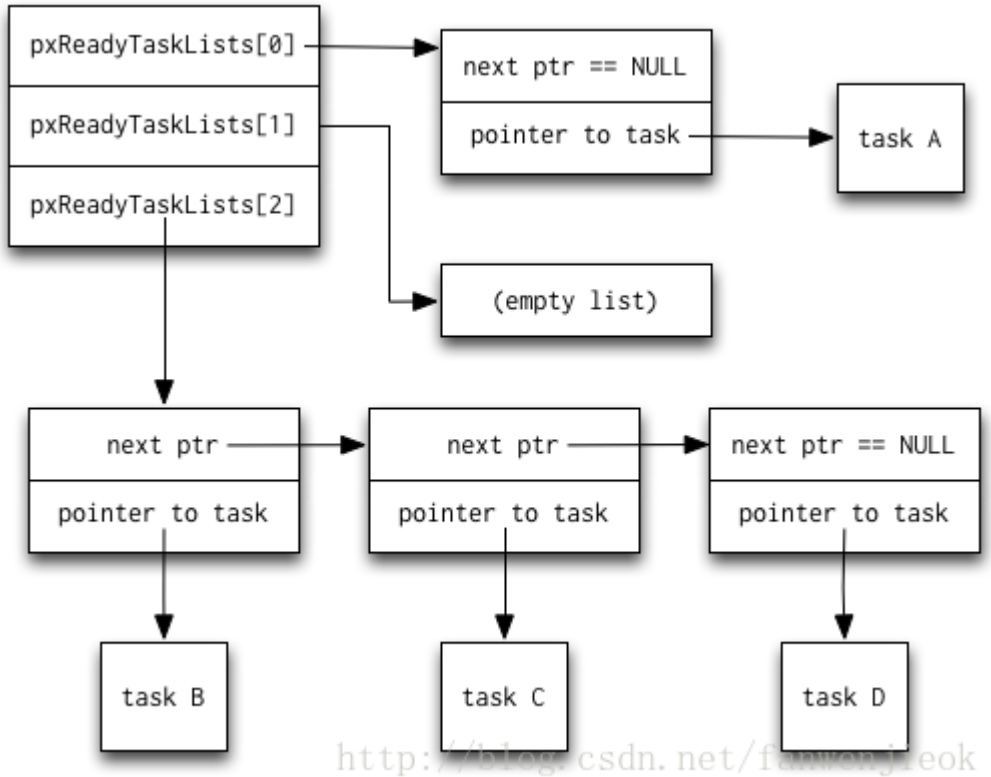


图3.2：FreeRTOS的就绪列表的基本视图

现在我们有了大致概述的方式，让我们去深究它的细节。我们将着眼于三个主要FreeRTOS的**数据结构**：任务，列表和队列。

3.4. 任务

所有操作系统的主要工作是运行和协调用户任务。像多数操作系统一样，FreeRTOS中的基本工作单元是任务。FreeRTOS的使用任务控制块（TCB）来表示每个任务。

任务控制块（TCB）

TCB的在tasks.c定义是这样的：

```cpp
typedef struct tskTaskControlBlock
{
  volatile portSTACK_TYPE *pxTopOfStack;                /* Points to the location of
                                                           the last item placed on
                                                           the tasks stack.  THIS
                                                           MUST BE THE FIRST MEMBER
                                                           OF THE STRUCT. */


  xListItem    xGenericListItem;                        /* List item used to place
                                                           the TCB in ready and
                                                           blocked queues. */
  xListItem    xEventListItem;                          /* List item used to place
                                                           the TCB in event lists.*/
  unsigned portBASE_TYPE uxPriority;                    /* The priority of the task
                                                           where 0 is the lowest
                                                           priority. */
  portSTACK_TYPE *pxStack;                              /* Points to the start of
                                                           the stack. */
  signed char    pcTaskName[ configMAX_TASK_NAME_LEN ];  /* Descriptive name given
```

```
                                                    to the task when created.
                                                    Facilitates debugging
                                                    only. */


  #if ( portSTACK_GROWTH > 0 )
    portSTACK_TYPE *pxEndOfStack;                   /* Used for stack overflow
                                                    checking on architectures
                                                    where the stack grows up
                                                    from low memory. */
  #endif


  #if ( configUSE_MUTEXES == 1 )
    unsigned portBASE_TYPE uxBasePriority;          /* The priority last
                                                    assigned to the task -
                                                    used by the priority
                                                    inheritance mechanism. */
  #endif


} tskTCB;
```

TCB在pxStack里存储堆栈的起始地址，以及在pxTopOfStack里存储当前堆栈的顶部。如果堆栈“向上”增长到更高的地址，它还在pxEndOfStack存储堆栈的结束的指针来检查堆栈溢出。如果堆栈“向下”增长到更低的地址，那么通过比较当前堆栈的顶部与pxStack中的堆内存起始位置来检查溢出。

TCB在uxPriority和uxBasePriority中存储任务的初始优先级。一个任务在它创建的时候被赋予优先级，同时任务的优先级是可以被改变的。如果FreeRTOS实现了优先级继承，那么当任务临时提升到“继承的”优先级时，它使用uxBasePriority去记住原来的优先级。（优先级继承，请参见下面关于互斥的讨论。）

每个任务有两个清单项目给FreeRTOS操作系统的各种调度列表使用。当一个任务被插入到FreeRTOS的一个列表中，不会直接向TCB插入一个指针。取而代之的是，它向TCB的xGenericListItem或xEventListItem插入一个指针。这些xListItem变量，比起若是仅仅获得一个指向TCB的指针来说，让FreeRTOS的列表变得更加灵活。

任务可以在以下四种状态之一：运行，准备运行，挂起或阻塞。你可能希望每个任务都有一个变量来告诉FreeRTOS它正处于什么状态，但事实上并非如此。相反，FreeRTOS通过把任务放入相应的列表：就绪列表，挂起列表等，隐式地跟踪任务状态。随着任务的变化，从一个状态到另一个，FreeRTOS的只是简单的将它从一个列表移动到另一个。

任务设置

我们已经触及如何利用pxReadyTasksLists数组来选择和调度一个任务的；现在让我们来看一看一个任务最初是如何被创建的。当xTaskCreate()函数被调用的时候，一个任务被创建。FreeRTOS为一个任务新分配一个TCB对象，来记录它的名称，优先级，和其他细节，接着分配用户请求的总的堆栈（假设有足够使用的内存）和在TCB的pxStack成员中记录堆内存的开始。

堆栈被初始化看起来就像一个已经在运行的新任务被上下文切换所中断。这就是任务调度处理最新创建的任务的方法，同样也是处理运行了一段时间的任务的方法；任务调度在不需要任何特殊（case）代码的情况下去处理新的任务。

任务的堆栈建立看起来像是它通过上下文切换来被中断，这个方法是取决于FreeRTOS正在运行的架构，但这个ARM Cortex-M3处理器的实现是一个很好的例子：

```
[cpp]

unsigned int *pxPortInitialiseStack( unsigned int *pxTopOfStack,
                                     pdTASK_CODE pxCode,
                                     void *pvParameters )
{
```

```
    /* Simulate the stack frame as it would be created by a context switch interrupt. */
    pxTopOfStack--; /* Offset added to account for the way the MCU uses the stack on
                       entry/exit of interrupts. */
    *pxTopOfStack = portINITIAL_XPSR;  /* xPSR */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) pxCode;  /* PC */
    pxTopOfStack--;
    *pxTopOfStack = 0;  /* LR */
    pxTopOfStack -= 5;  /* R12, R3, R2 and R1. */
    *pxTopOfStack = ( portSTACK_TYPE ) pvParameters;  /* R0 */
    pxTopOfStack -= 8;  /* R11, R10, R9, R8, R7, R6, R5 and R4. */

    return pxTopOfStack;
}
```

当一个任务被中断的时候，ARM Cortex-M3处理器就压寄存器入堆栈。pxPortInitialiseStack()修改堆栈使之看来像是即便任务实际上还未开始运行，寄存器就已经被压入了。已知的值被存储到堆栈，赋给ARM寄存器xPSR，PC，LR，和R0。剩余的寄存器R1-R12获得由栈顶指针递减分配给它们的寄存器空间，但没有具体的数据存储在这些寄存器堆栈内。ARM架构告诉我们那些寄存器在复位的时候未被定义，所以一个（非弱智的）程序将不依赖于已知的值。

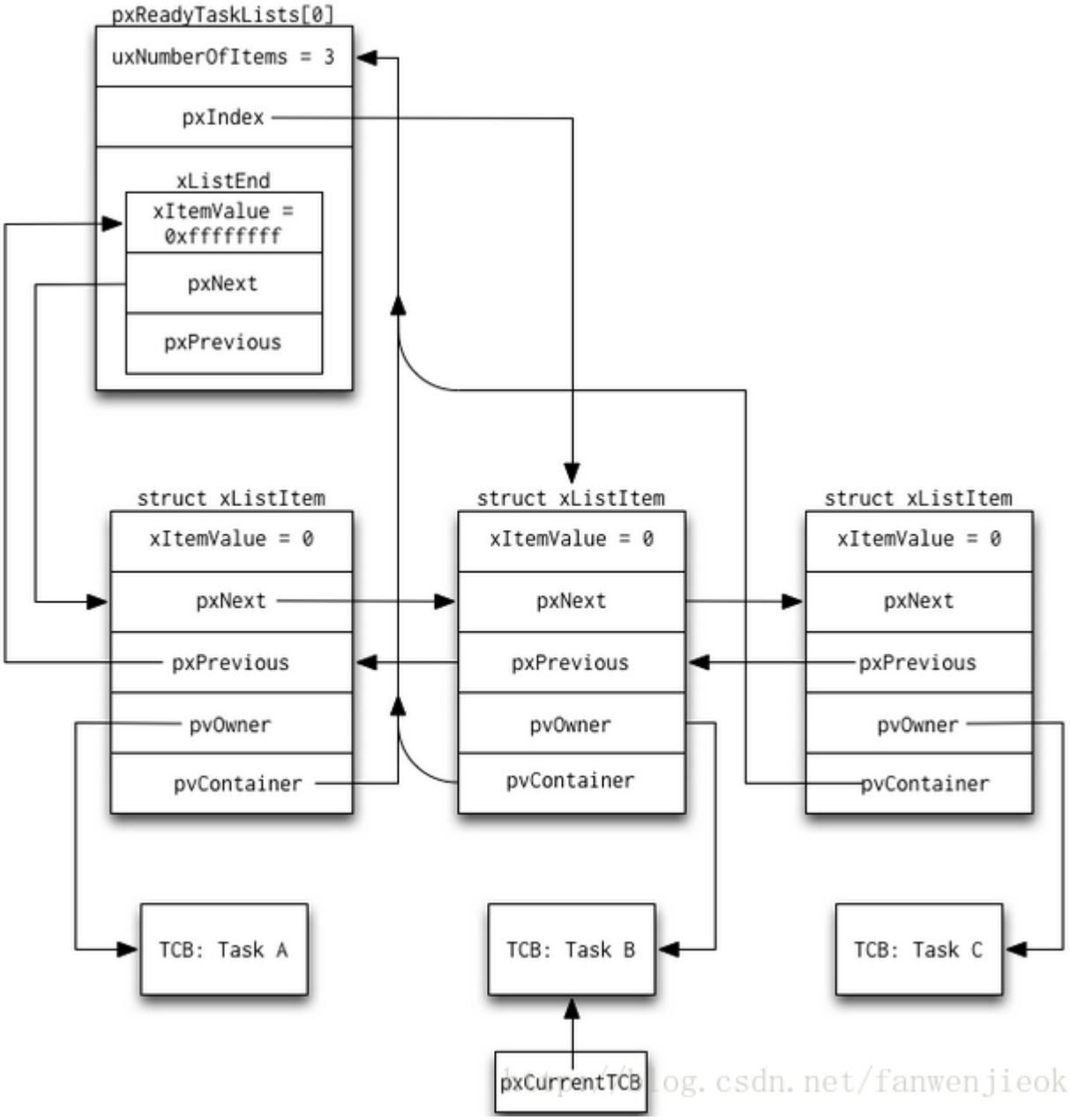
堆栈准备好后，任务几乎是同时准备运行。首先，FreeRTOS禁用中断：我们将开始使用就绪列表和其他任务调度的数据结构，同时我们不希望它们被其他人背着我们私底下修改。

如果这是被创建的第一个任务，FreeRTOS将初始化调度的任务列表。FreeRTOS操作系统的调度有一个就绪列表的数组，pxReadyTasksLists []，为每一个可能的优先级提供一个就绪列表。FreeRTOS也有一些其他的列表用来跟踪任务的挂起，终止和延时。现在这些也都是初始化的。

任何第一次初始化完成后，新的任务以它指定的优先级被加入就绪列表。中断被重新启用，同时新任务的创建完成。

3.5. 列表

任务之后，最常用的FreeRTOS数据结构是列表。FreeRTOS使用列表结构来跟踪调度任务，并执行队列。



这个FreeRTOS的列表是一个有着几个有趣的补充的标准循环双链表。下面就是列表元素：

```
[cpp]

struct xLIST_ITEM
{
  portTickType xItemValue;                  /* The value being listed.  In most cases
                                               this is used to sort the list in
                                               descending order. */
  volatile struct xLIST_ITEM * pxNext;      /* Pointer to the next xListItem in the
                                               list.  */
  volatile struct xLIST_ITEM * pxPrevious;  /* Pointer to the previous xListItem in
                                               the list. */
  void * pvOwner;                           /* Pointer to the object (normally a TCB)
                                               that contains the list item.  There is
                                               therefore a two-way link between the
                                               object containing the list item and
                                               the list item itself. */
```

```
  void * pvContainer;                    /* Pointer to the list in which this list
                                            item is placed (if any). */
};
```

每个元素持有一个数字，xItemValue，这通常是一个被跟踪的任务优先级或者是一个调度事件的计时器值。列表保存从高到低的优先级指令，这意味着最高的优先级xItemValue（最大数）在列表的最前端，而最低的优先级xItemValue（最小数）在列表的末尾。
pxNext和pxPrevious指针是标准链表指针。pvOwner 列表元素所有者的指针。这通常是任务的TCB对象的指针。
pvOwner被用来在vTaskSwitchContext()中加快任务切换：当最高优先级任务元素在pxReadyTasksLists[]中被发现，这个列表元素的pvOwner指针直接连接到需要任务调度的TCB。
pvContainer指向自己所在的这个列表。若列表项处于一个特定列表它被用作快速终止。任意列表元素可以被置于一个列表，如下所定义：

```cpp
typedef struct xLIST
{
  volatile unsigned portBASE_TYPE uxNumberOfItems;
  volatile xListItem * pxIndex;          /* Used to walk through the list.  Points to
                                            the last item returned by a call to
                                            pvListGetOwnerOfNextEntry (). */
  volatile xMiniListItem xListEnd;       /* List item that contains the maximum
                                            possible item value, meaning it is always
                                            at the end of the list and is therefore
                                            used as a marker. */
} xList;
```

列表的大小任何时候都是被存储在uxNumberOfItems中，用于快速列表大小操作。所有的列表都被初始化为容纳一个元素：xListEnd元素。xListEnd.xItemValue是一个定点值，当portTickType是16位数时，它等于xItemValue变量的最大值：0xffff，portTickType是32位数时为0xffffffff。其他的列表元素也可以使用相同的值；插入算法保证了xListEnd总是列表项中最后一个值。
自列表从高到低排序后，xListEnd被用作列表开始的记号。并且，自循环开始，xListEnd也被用作列表结束的记号。
你也许可以用一个单独的for()循环或者是函数调用来访问大多数“传统的”列表，去做所有的工作，就像这样：

```cpp
for (listPtr = listStart; listPtr != NULL; listPtr = listPtr->next) {
  // Do something with listPtr here...
}
```

并且返回pxIndex。（当然它也会正确检测列尾环绕。）这种，当执行遍历的时候使用pxIndex，由列表自己负责跟踪“在哪里”的方法，使FreeRTOS可以休息而不用关心这方面的事。

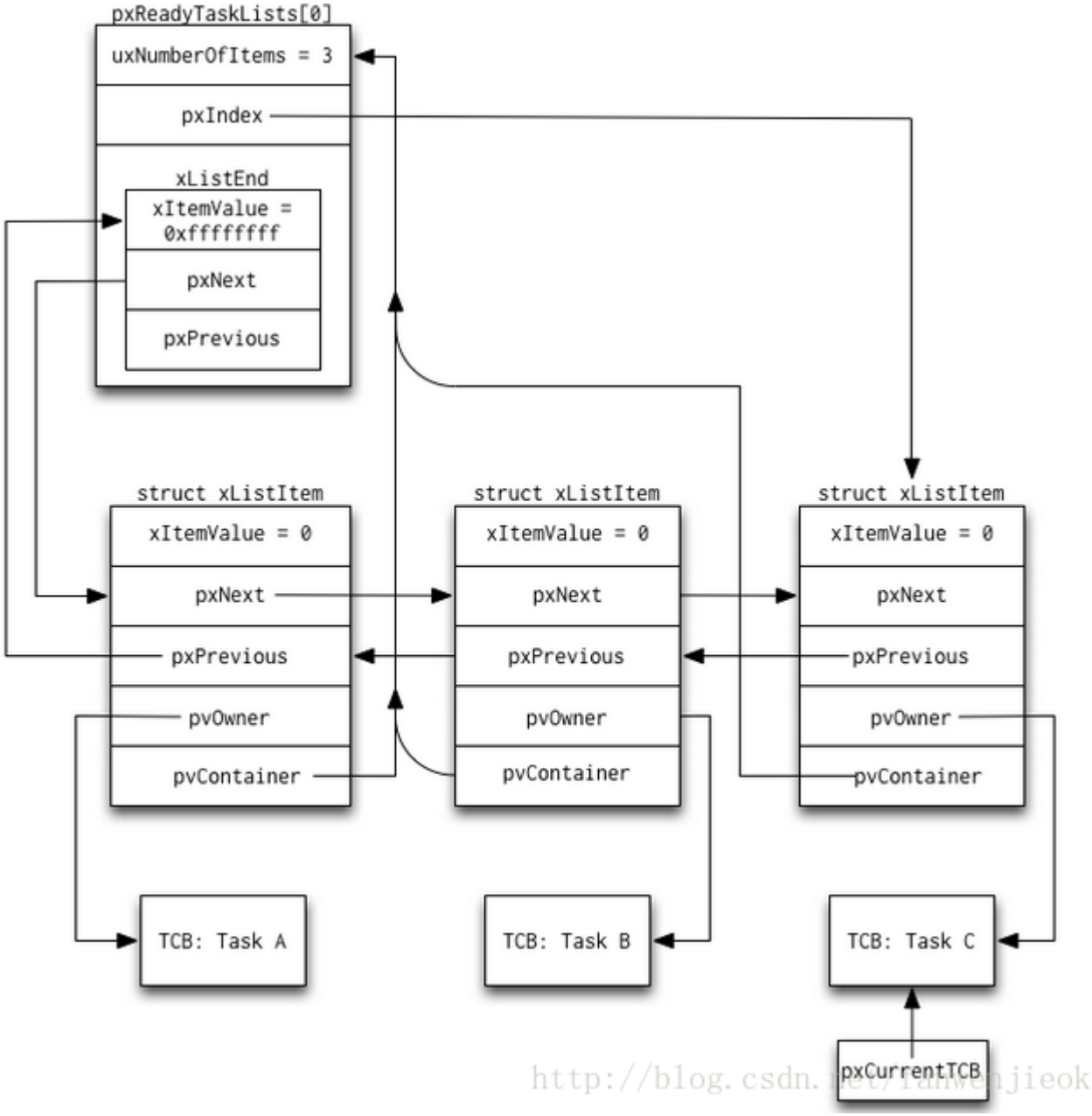


图3.4：系统节拍计时器下的FreeRTOS就绪列表全貌

FreeRTOS经常需要通过多个for()和while()循环，也包括函数调用来访问列表，因此它使用操纵pxIndex指针的列表函数来遍历这个列表。这个列表函数listGET_OWNER_OF_NEXT_ENTRY()执行pxIndex = pxIndex->pxNext;pxReadyTasksLists[]列出了在vTaskSwitchContext()中已经操纵完成的内容，是如何使用pxIndex的一个很好的例子。让我们假设我们仅有一个优先级，优先级0，并且有三个任务在此优先级上。这与我们之前看到的基本就绪列表图相似，但这一次我们将包括所有的数据结构和字段。

就如你在图3.3中所见，pxCurrentTCB显示我们当前正在运行任务B。下一个时刻，vTaskSwitchContext()运行，它调用listGET_OWNER_OF_NEXT_ENTRY()载入下一个任务来运行。如图3.4所示，这个函数使用pxIndex->pxNext找出下一个任务是任务C，并且pxIndex指向任务C的列表元素，同时pxCurrentTCB指向任务C的TCB。

请注意，每个struct xlistitem对象实际上都是来自相关TCB的xGenericListItem对象。

3.6. 队列

FreeRTOS允许任务使用队列来互相间通信和同步。中断服务程序（ISRs）同样使用队列来通信和同步。

基本队列数据结构如下：

```
[cpp]

typedef struct QueueDefinition
{
```

```
    signed char *pcHead;                    /* Points to the beginning of the queue
                                               storage area. */
    signed char *pcTail;                    /* Points to the byte at the end of the
                                               queue storage area. One more byte is
                                               allocated than necessary to store the
                                               queue items; this is used as a marker. */
    signed char *pcWriteTo;                 /* Points to the free next place in the
                                               storage area. */
    signed char *pcReadFrom;                /* Points to the last place that a queued
                                               item was read from. */


    xList xTasksWaitingToSend;              /* List of tasks that are blocked waiting
                                               to post onto this queue.  Stored in
                                               priority order. */
    xList xTasksWaitingToReceive;           /* List of tasks that are blocked waiting
                                               to read from this queue. Stored in
                                               priority order. */


    volatile unsigned portBASE_TYPE uxMessagesWaiting;  /* The number of items currently
                                                           in the queue. */
    unsigned portBASE_TYPE uxLength;                    /* The length of the queue
                                                           defined as the number of
                                                           items it will hold, not the
                                                           number of bytes. */
    unsigned portBASE_TYPE uxItemSize;                  /* The size of each items that
                                                           the queue will hold. */


} xQUEUE;
```

这是一个颇为标准的队列，不但包括了头部和尾部指针，而且指针指向我们刚刚读过或者写过的位置。

当刚刚创建一个队列，用户指定了队列的长度和需要队列跟踪的项目大小。pcHead和pcTail被用来跟踪队列的内部存储器。加入一个项目到队列就对队列内部存储器进行一次深拷贝。

FreeRTOS用深拷贝替代在项目中存放一个指针是因为有可能项目插入的生命周期要比队列的生命周期短。例如，试想一个简单的整数队列使用局部变量，跨几个函数调用的插入和删除。如果这个队列在局部变量里存储这些整数的指针，当整数的局部变量离开作用域时指针将会失效，同时局部变量的存储空间将被新的数值使用。

什么需要用户选择使用队列。若内容很少，用户可以把复制的内容进行排列，就像上图中简单整数的例子，或者，若内容很多，用户可以排列内容的指针。请注意，在这两种情况下FreeRTOS都是在做深拷贝：如果用户选择排列复制的内容，那么这个队列存储了每项内容的一份深拷贝；如果用户选择了排列指针，队列存储了指针的一份深拷贝。当然，用户在队列里存储了指针，那么用户有责任管理与内存相关的指针。队列并不关心你存储了什么样的数据，它只需要知道数据的大小。

FreeRTOS支持阻塞和非阻塞队列的插入和移除。非阻塞队列操作会立即返回"队列的插入是否完成?"或者 "队列的移除是否完成？"的状态。阻塞操作则根据特定的超时。一个任务可以无限期地阻塞或者在有限时间里阻塞。

一个阻塞任务——叫它任务A——将保持阻塞只要它的插入/移除操作没有完成，并且它的超时（如果存在）没有过期。如果一个中断或者另一个任务编辑了这个队列以便任务A的操作能够完成，任务A将被解除阻塞。如果此时任务A的队列操作仍然是允许的，那么它实际上会执行操作，于是任务A会完成它的队列操作，并且返回“成功”的状态。不过，任务A正在执行的那个时间，有可能同时有一个高优先级任务或者中断也在同一个队列上执行另一个操作，这会阻止任务A正在执行的操作。在这种情况下任务A将检查它的超时，同时，如果它未超时就恢复阻塞，否则就返回队列操作“失败”的状态。

特别需要注意的是，当任务被阻塞在一个队列时，系统保持运行所带来的风险；以及当任务被阻塞在一个队列时，有其他任务或中断在继续运行。这种阻塞任务的方法能不浪费CPU周期，使其他任务和中断可以有效地使用CPU周

期。

FreeRTOS使用xTasksWaitingToSend列表来保持对正阻塞在插入队列里的任务的跟踪。每当有一个元素被移出队列，xTasksWaitingToSend列表就会被检查。如果有个任务在那个列表中等待，那个是未阻塞任务。同样的，xTasksWaitingToReceive保持对那些正阻塞在移除队列里的任务的跟踪。每当有一个新元素被插入到队列，xTasksWaitingToReceive列表就会被检查。如果有个任务在那个列表中等待，那个是未阻塞任务。

信号灯和互斥

FreeRTOS使用它的队列与任务通信，也在任务间通信。FreeRTOS也使用它的队列来实现信号灯与互斥。

有什么区别?

信号灯与互斥听上去像一回事，但它们不是。FreeRTOS同样地实现了它们，但本来它们以不同的方式被使用。它们是如何不同地被使用？嵌入式系统宗师Michael Barr说这是在他文章中写得最好的，“信号灯与互斥揭秘”：

The correct use of a semaphore is for signaling from one task to another. A mutex is meant to be taken and released, always in that order, by each task that uses the shared resource it protects. By contrast, tasks that use semaphores either signal ["send" in FreeRTOS terms] or wait ["receive" in FreeRTOS terms] - not both.

正确使用的一个信号是从一个任务向另一个发信号。从每个使用被保护共享资源的任务来看，总是认为，一个互斥意味着获得和释放。相比之下，使用信号灯的任务不是发信号[在FreeRTOS里“发送”]就是在等信号[在FreeRTOS里“接收”]——不能同时。

互斥被用来保护共享资源。一个使用共享资源的任务获得互斥，接着释放互斥。当有另一个任务占据互斥时，没有一个任务可以获得这个互斥。这就是保证，在同一时刻仅有一个任务可以使用共享资源。

一个任务向另一个任务发信号时使用信号灯。以下引用Barr的文章：

For example, Task 1 may contain code to post (i.e., signal or increment) a particular semaphore when the "power" button is pressed and Task 2, which wakes the display, pends on that same semaphore. In this scenario, one task is the producer of the event signal; the other the consumer.

举例来说，任务一可能包含当“电源”按钮被按下时，发布（即，发信号或增加信号量）一个特定的信号灯的代码，并且唤醒显示屏的任务二，取决于同一个信号灯。在这种情况下，一个任务是发信号事件的制造者；另一个是消费者。

如果你是在信号灯和互斥有任何疑问，请查阅Michael的文章。

实现

FreeRTOS像队列一样来实现一个N元素的信号灯，这样就可以控制N个项。它没有去存储队列每项的任何实际数据；信号灯只关心有多少在队列的uxMessagesWaiting字段中被跟踪的队列记录，目前正被占用。当FreeRTOS的头文件semphr.h调用它的时候，它正在做“纯同步”。因此这个队列有一个零字节的项 (uxItemSize == 0)。每个信号灯uxMessagesWaiting字段递增或递减；没有项或数据的复制是需要的。

同信号灯一样，互斥也被实现为一个队列，不过有几个xQUEUE结构字段被用#defines重载：

```
/* Effectively make a union out of the xQUEUE structure. */
#define uxQueueType         pcHead
#define pxMutexHolder       pcTail
```

当互斥没有在队列中存储任何数据时，它不需要任何内部存储器，同样pcHead和pcTail字段也不需要。FreeRTOS设置uxQueueType字段（实际上的pcHead字段）为0来表明，这个队列正在被一个互斥使用。FreeRTOS使用重载的pcTail字段来实现互斥的优先级继承。

万一你不熟悉优先级继承，我将再次引用Michael Barr的话来定义它，这次是来自他的文章，“优先级倒置”：

[Priority inheritance] mandates that a lower-priority task inherit the priority of any higher-priority task pending on a resource they share. This priority change should take place as soon as the high-priority task begins to pend; it should end when the resource is released.

[优先级继承]要求低优先级任务继承任何高优先级任务的优先级，取决于它们共享的资源。这个优先级的改变应当在高优先级任务一开始挂起时就发生；资源被释放时就结束。

FreeRTOS使用pxMutexHolder字段（实际上是#define重载的pcTail字段）来实现优先级继承。FreeRTOS记录在pxMutexHolder字段里包含一个互斥的任务。当一个高优先级任务正在等待一个由低优先级任务取得的互斥，FreeRTOS“更新”低优先级任务到高优先级任务的优先级，直至这个互斥再次可用。

3.7. 结论

我们完成了对FreeRTOS架构的一览。希望你现在对于FreeRTOS的任务是如何执行以及通信有一个好的感觉。如果之前你从未了解过操作系统的内在，我希望现在你对于它们是如何工作的有一个基本的思路。

显然，本章没有覆盖FreeRTOS架构的全部。值得注意的是，我没有提到的内存分配，中断服务，调试，或MPU支持。本章也没有讨论如何设置或使用FreeRTOS。Richard Barry已经写了一本极好的书，使用FreeRTOS实时内核：实用指南，这本书就是讲这些内容的；如果你要使用FreeRTOS的话，我强烈推荐它。

3.8. 致谢

我想感谢Richard Barry创造和维护了FreeRTOS，并选择将它开源。在写这一章的时候Richard给予了极大的帮助，提供了关于FreeRTOS的历史以及非常有价值的技术回顾。

也感谢Amy Brown和Greg Wilson共同拉动这整个AOSA的事情。

最后也是最（最多），感谢我的妻子Sarah共同承担了我对本章的研究和写作。幸运的是，她在嫁给我的时候就知道我是一名极客。