**浅析嵌入式开发时选择实时操作系统的几大要点**

　　　　对许多嵌入式项目来说，系统设计师都倾向于选择实时操作系统(RTOS)。但RTOS总是必要的吗？答案是取决于具体的应用，因此了解我们要达到什么目标是决定RTOS是必要的还是花瓶的关键。

　　一般来说，在采用非实时操作系统(non-RTOS)的任何场合，也都可采用RTOS。但是，要找到一款具有完全相同应用编程接口(API)的匹配RTOS就相当困难了。因此，许多传统的操作系统(OS)在其内嵌入了一个RTOS。例如，Lynux-Works LynxOS和Bluecat Linux共享一个Linux API。LynxOS是一款硬RTOS，而Bluecat是Linux的一个衍生产品。

　　Linux继续在努力改善其实时性能，但其最长中断时延仍无法满足对RTOS来说至关重要的硬(hard)实时要求。这些问题最后都会归结为服务质量(QoS)。像RTLinux Free这样的平台补充了Linux，因为它们可提供硬实时级别的QoS。

　　要指出的很重要一点是：这类补充常常是在原始OS上集成一个RTOS编程环境。与传统台式或服务器OS相比，RTOS通常要小很多。RTOS常常针对更小和资源有限的MCU。例如，CMX的CMX-RTX和CMX-Tiny+可运行在8位MCU到64位处理器上。

　　8位处理器不断增加的计算能力和存储容量正使得RTOS对这些平台具有更大吸引力。但是，通常16位或以上平台才需要OS或RTOS，常见的RTOS选择有Express Logic的ThreadX、Wind River的VxWorks、Micrium的uCOS-II、以及Green Hills Software的velOSity。取决于需求，MontaVista的Linux可在几个微秒的水平上满足16位和32位平台的要求。

**RTOS核心：调度和分割**

　　大多数程序员不熟悉RTOS的限制和要求。大多数人通常因其性能选择RTOS。大多数RTOS产品代码少和速度快，现在RTOS还提升了一致性。RTOS除能很快完成任务外，还能保证很好地完成任务。

　　在许多应用中，一个迟到的结果可以是灾难性的。因此，人们宁愿在一个要求的时限内获得较差的结果。这些应用通常被称为硬实时系统。硬实时不是指系统响应有多快或多快一个系统能响应，而是指系统能多可靠地满足特定的要求。

　　一个硬实时系统可能有一个一分钟的固定周期时间，它要求的响应时间为一秒。理论上，这样的要求几乎所有的操作系统都能实现。但事实并非总是如此，正如任何一个人都能证明等待台式计算机应用在一分钟之内做出响应需要等多久。

　　硬实时系统通常具有更短的周期时间和更紧苛的响应要求。更快的处理器速度总是有帮助的，多内核平台也能改善响应速度。对开发人员来说，窍门在于把系统需求与硬件和软件匹配起来，然后才是RTOS在嵌入式应用中的重要性。

　　一个RTOS可以实现一系列调度策略，但应用经常会制约一个程序员的选择。非优先式调度(non-preemptive scheduling)的实现虽不重要，但在一些应用中很有用。另一方面，任务内的非优先式调度可在优先式系统的顶部实现。

　　不应该轻忽非优先式调度，特别在新型多内核处理器出现以后。这里，硬件可被调整到处理一个基于事件的操作，其中线程将等待外部事件的发生。对处理多线程的单核处理器来说，该方法一般不适用。但对有许多内核的多核系统说，典型情况是为一个外设指定一个核。所以，在等待事件发生期间，使该核空闲起来是有意义的。

　　其结果是，优先式、中断驱动的RTOS架构占据了业已部署的大部分平台。这些平台有一系列的要求、问题和解决方案（见图）。虽然借助硬件手段(多个寄存器组合、硬件调度、任务切换、以及分层中断优先级系统等)可显着缩短中断时延，但该时延永远是一个问题。

　　优先式处理会带来若干问题。它们大多是与时序关联的，如竞争条件、死循环、空耗等待和优先级转换，它们发生在低优先级任务A拥有更高优先级任务B的同步资源，而优先级比A高的任务C正在运行。

　　如果没有像优先级置顶(priority ceilings)这样的特性，任务C就可以阻止任务A和任务C运行。优先级置顶特性可以把任务A的优先级改变成与任务C的优先级一样，从而允许任务A运行并最终释放任务C所需的资源。至此，任务A的优先级复原，任务C就可以继续运行。

　　程序员必须解决的其它与时序相关的问题通常是难以定位和纠正的缺陷源。在定位这些缺陷时跟踪工具就变成了很有价值的手段，因为诸如受阻的任务等症候是这些问题的唯一表现形式。

　　就操作系统所需的特性来看，重入库(reentrant library)特性在RTOS环境下是可有可无的。但在一个典型的操作系统中，由于任务和程序常常是随机的和变化的，而且常公用库，因此重入库是一个必须的特性。

　　在嵌入式环境中，对在系统中运行的程序和任务一般会有更多的控制要求。通常，除操作系统接口(可以是重入也可能是非重入的)外，各任务从不共享任何代码。程序员(特别是那些负责设备驱动程序的)需要注意这一重入性问题。

　　现在业内已有很多的任务同步机制，从互斥(mutex)到消息系统。从RTOS的角度，这些机制在诸如竞争条件此类的同步问题上，没有什么差异。

　　在MCU和操作系统中，定时器很常见。至少，一个定时器可被用作时钟。但由于定时器是如此的有用，以至于它常以一种特殊方式实现出来。POSIX规范甚至把定时器定义为组件。定时器还可当作看门狗来用。

　　在许多MCU中，一个定时器可以设置用来唤醒处在休眠模式的系统。一些实现允许操作系统把其用作一个通用定时器，尽管这一唤醒特性独立于操作系统。

　　一些系统具有带不同特性的多种定时器来满足不同的要求。一些定时器可被同步用以为电机控制应用提供同时的脉宽调制(PWM)流。对RTOS来说，一个定时器通常可用以实现时钟和提供时间切片支持。

　　定时器也支持时间切片。时间切片常见于时间共享系统，它给每种应用一个合理的时间片断来执行。可在任一中断层级上实现这种轮询调度。

　　通常，由时钟提供的时间切片是固定时长的，每个任务在获得优先权前将被给予同样长度的时间切片来执行。当然，该策略是随机的且可有多种实现。例如，可变的时间切片宽度将允许时间以每个任务为单位进行分配，其中一些任务获得的时间会比另一些长；而若采用任务优先级方法，则有可能使低优先级任务得不到响应。

　　许多RTOS采用固定调度器。其它RTOS则允许替换或定制，但RTOS中的另一部分支持各种策略。这一灵活方法使得像Linux这样的操作系统能够提供实时支持，与此同时，它们还能在时间切片环境下运行多种应用。实时任务具有高优先级，且在一般用户任务前得到执行。

　　Linux实际上具有一个更复杂的调度系统，它对任务是通过轮询方法把控制权转交给具有相同优先级的其它任务还是一直运行到结束做出了具体约定。像Open Kernel Labs的OKL4虚拟化RTOS平台解决了该问题。

**基本通信**

　　一些文献把任务同步和通信分开来说，但总的来说，它们是一回事。实际上就是讲信息是如何交换的。基于消息传递的RTOS最清楚地体现出这点。这里，消息系统处理所有通信且不区分通信和同步。

　　至少，RTOS必须提供一个相互排斥的本原，如互斥。其它东西可构建在该本原上。在许多场合，如消息传递系统，对相互排斥的支持隐藏在操作系统内。只有更高级别的消息功能显露于外。

　　消息系统有各种名称，从管道到队列。其实现可横跨从单处理器、单存储器模式到多内核群集系统。Enea的OSE RTOS和QNX的Neutrino是基于消息传递的两个主线RTOS。

　　不管选择了什么方法或API，通信系统必须在某一程度上被整合进操作系统。因此，若主动队列中的任务必须等待一个事件，则该任务可被移走。类似，引发一个事件从而导致另一个任务活动的任务将产生一个调度行为。

　　通信、事件和调度可与硬件关联起来，这是RTOS必须处理的其它一些事。TI的DSP/BIOS是一款RTOS，它设计用于运行在像TI的DaVinci双核系统的DSP上。DSP/BIOS的一个主要功能是处理 ARM 核和DSP 核间的通信。

　　向更多大内核的发展将很可能会保留RTOS或OS。不过，小内核阻止或限制了采用RTOS的可能性。Intellasys的SEAforth 40C18芯片带有40个运行Forth的小型18位内核。指令很精简，每个字包含四条指令。

　　每个内核有64个字的 ROM和RAM，该芯片只能容纳10,000指令。当然，这只够装下一个程序，安装RTOS是不可能的。不过，整个芯片上有足够空间安装一个操作环境的特定部分。同样，适于该平台的应用常常是特定的。于是，由于硬件可处理内核之间通信和任务调度，因此RTOS类的支持并不需要。

**资源管理**

　　使RTOS脱颖而出的是其管理资源(包括时间和存储器)的能力。时序问题与中断响应时间有关，但资源管理时序问题也会出现。虽然中断解决了一系列时序问题，但各应用仍必须利用资源。

　　考虑存储器分配情况。许多实时应用不采用动态存储器分配，以确保存储器分配和回收时所产生的不同不会变成一个问题。需要动态存储器分配的应用常把存储器划分为实时和非实时。后者处理动态存储器分配。典型情况下，在使用前，实时部分必须被分配有足够的存储器。

　　在实时嵌入式应用中采用C和C++是因为存储器和其它资源的用法是显式的。实时任务需要避免采用C和C++。特别是，当存储器分配和回收更容易隐藏时采用C++是很困难的。

　　像Java和C#这样的语言带来的挑战更大，它们与生俱来地采用动态存储器分配。程序员可控制存储器分配和回收。在某些情况下，编程环境可以强化存储器分配和回收。

　　Java实时规范(RTSJ)定义了创建不需要垃圾回收的Java应用的方法。RTSJ是在Java框架内这样做的，从而使程序员在不被存储器分配限制的条件下享有Java的好处。

　　Sun和DDC-I都实现了RTSJ。DDC-I的实现支持x86和PowerPC平台。Aonix有一个称为PERC的类似平台。这些平台以实时、同时的垃圾回收为特征，从而使在不受存储器分配限制的情况下，在Java内编写实时应用成为可能。

但因系统必须允许线程为垃圾回收器进行转换，所以实时要求并非那么紧迫。另一方面，垃圾回收器将耗费时序资源，所以，只有实时任务方可保证满足一定的期限要求。快是好事，但及时才是RTOS的天条。

　　考察实时平台时，考虑之一是存储器分配对系统的整体影响。许多系统可工作在从不改变的静态分配环境，但更多的动态系统可从实时垃圾回收中获益。研究表明，垃圾回收的效益与确定的存储器分配是可比的。

　　围绕诸如Java和C#等虚拟机类型平台的另一个问题是对just-in-time(JIT)编译器的使用限制。基于这些系统的实时系统必须采用类似C和C++等所用的提前（ahead-of time，AOT）编译器。

　　设计师会因其更高的生产力、更低的出错率以及安全性等特点选用Java 或C#。所以，对制定一个称为 JSR-302的用于对安全有至高要求应用的Java规范就不足为奇了。

**保护RTOS**

　　RTOS受到其运行的硬件平台的限制。可对缺少存储器保护的硬件加以保护，但安全级别会受到限制。但存储器和虚拟机可以更高水平的安全性支持引导。诸如SE Linux、Green Hills Integrity和 LynuxWorks LynxSecure Embedded Hypervisor以及 LynxOS-SE RTOS内的安全策略可比典型RTOS提供可靠得多的保护。但成本也高，所以开发者需对此进行权衡。

　　实时系统开发者不得不应对策略实现和边界问题。取决于信息的来所去处，安全支持会花很长时间。正是为此引入了分区系统，所以，可在边界采取安全措施且把应用的非实时部分放在这部分空间内。

**可感知OS的调试器**

　　当考虑选用操作系统时，对调试器的支持是个关键。这种支持体现在两个方面：内核和设备驱动器调试以及操作系统感知。

　　内核调试对设备驱动器的创建和支持以及内核强化很重要。在许多情况，为处理RTOS的内核，需要专用调试器。它也要求能理解内核环境以及应用环境。

　　OS感知可更深入地了解操作系统。支持方式可以是从提供有关OS服务状态的信息到调整任务调度等方方面面。同样，能感知OS的调试器可在停止其它应用或线程的同时允许其它应用或线程的运行。