嵌入式软件设计中查找缺陷的几个技巧

结构测试或白盒测试能有效地发现代码中的逻辑、控制流、计算和数据错误。这项测试要求对软件的内部工作能够一览无遗(因此称为“白盒”或“玻璃盒”)，以便了解软件结构的详细情况。它检查每个条件表达式、数学操作、输入和输出。由于需要测试的细节众多，结构测试每次检查一个软件单元，通常为一个函数或类。

代码审查也使用与实现缺陷和潜在问题查找同样复杂的技术。与白盒测试一样，审查通常针对软件的各个单元进行，因为一个有效的审查过程要求的是集中而详尽的检查。

与审查和白盒测试不同，功能测试或黑盒测试假设对软件的实现一无所知，它测试由受控输入所驱动的输出。功能测试由测试人员或开发人员所编写的测试过程组成，它们规定了一组特定程序输入对应的预期程序输出。测试运行之后，测试人员将实际输出与预期输出进行比较，查找问题。黑盒测试可以有效地找出未能实现的需求、接口问题、性能问题和程序最常用功能中的错误。

虽然将这些技术结合起来可以找出隐藏在一个特定软件程序中的大部分错误，但它们也有局限。代码审查和白盒测试每次只针对一小部分代码，忽视了系统的其它部分。黑盒测试通常将系统作为一个整体来处理，忽视了实现的细节。一些重要的问题只有在集中考察它们在整个系统内相互作用时的细节才能被发现；传统的方法无法可靠地找出这些问题。必须整体地检查软件系统，查找具体问题的特定原因。由于详尽彻底地分析程序中的每个细节和它与代码中所有其它部分之间的相互作用通常是不大可能的，因此分析应该针对程序中已经知道可能导致问题的特定方面。本文将探讨其中三个潜在的问题领域：

* 堆栈溢出

* 竞争条件

* 死锁

读者可在网上阅读本文的第二部分，它将探讨下列问题：

* 时序问题

* 可重入条件

在采用多任务实时设计技术的系统中，以上所有问题都相当普遍。

处理器使用堆栈来存储临时变量、向被调函数传递参数、保存线程“状态”，等等。如果系统不使用虚拟内存(换句话说，它不能将内存页面转移到磁盘上以释放内存空间供其它用途)，堆栈将固定为产品出厂时的大小。如果由于某种原

因堆栈越出了编程人员所分配的数量范围，程序将变得不确定。这种不稳定可能导致系统发生严重故障。因此，确保系统在最坏情况下能够分配到足够的堆栈至关重要。

确保永不发生堆栈溢出的唯一途径就是分析代码，确定程序在各种可能情况下的最大堆栈用量，然后检查是否分配了足够的堆栈。测试不大可能触发特定的瞬时输入组合进而导致系统出现最坏情况。

堆栈深度分析的概念比较简单：

1. 为每个独立的线程建立一棵调用树。

2. 确定调用树中每个函数的堆栈用量。

3. 检查每棵调用树，确定从树根到外部“树叶”的哪条调用路径需要使用的堆栈最多。

4. 将每个独立线程调用树的最大堆栈用量相加。

5. 确定每个中断优先级内各中断服务程序(ISR)的最大堆栈用量并计算其总和。但是，如果 ISR 本身没有堆栈而使用被中断线程的堆栈，则应将 ISR 使用的最大堆栈数加到各线程堆栈之上。

6. 对于每个优先级，加上中断发生时用来保存处理器状态的堆栈数。

7.如果使用 RTOS，则加上 RTOS 自身内部用途需要的最大堆栈数(与应用代码引发的系统调用不同，后者已包含在步骤 2 中)。

除此之外，还有两个重要事项需要考虑。首先，仅仅从高级语言源代码建立的调用树很可能并不完善。大部分编译器采用运行时库(run-time library)来优化常用计算任务，如大值整数的乘除、浮点运算等，这些调用只在编译器产生的汇编语言中才可见。运行时库函数本身可能使用大量的堆栈空间，在分析时必须将它们包括进去。如果使用的是 C++语言，则以下所有类型的函数(方法)也都必须包含到调用树内：结构器、析构器、重载运算符、复制结构器和转换函数。所有的函数指针也都必须进行解析，并且将它们调用的函数包含进分析之中。

第二，编译器使用一个 C 库来实现 memcpy()、cos()和 atof ()等标准函数，而这些例程的源代码可能无法得到。如果能够得到它们的源代码，就有可能确定程序用到的每个库调用在最坏情况下的堆栈使用数量。如果这些库只包含在目标文件中，则编译器厂商必须提供每个库例程使用的堆栈数。如果没有这些信息，就无法通过分析来确定最坏情况下程序使用的最大堆栈数。幸运的是，许多面向嵌入式系统的编译器厂商都提供这些信息。

通常，每次一个函数被调用时，编译器将使用堆栈来保存返回地址并传递函数参数。函数的自动(局部)变量通常也在堆栈当中。不过，由于编译器会尽可能

通过将参数或局部变量放入寄存器来优化代码，因此检查汇编语言以精确地确定堆栈用量非常重要。编译器也有可能在代码中的其它地方选择使用堆栈，如用堆栈来保存中间计算结果。

有些与编译器一起打包销售的开发环境包含生成调用树的工具，还有许多第三方的调用树生成工具。但是，除非它们能够对汇编语言进行分析，否则这些工具可能会遗漏运行时库和 C 库的调用。不过无论在哪种情况下，开发分析汇编语言文件并提取函数名称以及各函数内部调用的脚本都比较简单。分析的结果可写入一个文件，而这个文件能够方便地输入到表格之中。

确定了各个函数的堆栈用量之后，必须计算每个线程所需的最大堆栈数。由于一般程序通常涉及数百个函数，调用跨越多层深度，处理这些信息的一种简便方法就是采用分析表格。如表 1 所示，表格的各行包含了函数名称、该函数使用的最大堆栈数(包括调用其它函数所需的堆栈数)，以及它调用的所有函数的清单。通过编程控制，这个表格从每个函数的“根”开始迭代循环，计算该函数及其调用的所有函数需要的堆栈。这些信息存放在堆栈路径列中，这样，采用每个线程根函数(如 main)的堆栈路径数据就可以方便地计算出需要的最大堆栈数了。这个过程包含了先前介绍的堆栈分析过程中的前四个步骤。

有时候，采用堆栈深度分析过程可能是无法做到，或者是不实际的。如果无法得到运行时库或 C 库的源代码，而编译器厂商又没有提供任何堆栈使用信息，就不可能进行完整的堆栈分析。在这种情况下，有两种选择：

1. 在测试期间，观察堆栈所能达到的深度，并保证有较大的堆栈空间余量。

2. 检测堆栈溢出，并采取改进措施。

观察堆栈深度的方法很简单：

* 向整个内存堆栈区写入一个特定的数据图案符号，如 55AA。

* 在预期使用最大堆栈空间的条件下运行系统。

* 使用仿真器或其它工具检查堆栈存储区，看有多少符号图案由于堆栈的使用而被改写了。

当然，这些步骤并不能保证在一些不同条件下不会需要更多的堆栈，但确实可以表明所需要的最小堆栈数。

使用带内存管理单元(MMU)的处理器时，有可能检测出运行时的堆栈溢出现象。MMU 将内存划分为多个区域，用一个受保护的内存段来“警戒”堆栈区域。发生堆栈溢出时，处理器将访问这个受保护段。这个操作将引发一个异常事件(如产生 SIGSEGV 信号)，可被程序捕获到。创建线程时，与实时 POSIX 标准兼容的 RTOS 提供有这种堆栈警戒功能选项，大大简化了编程人员的工作。GNU 工具等其它开发环境包含有编译器开关，可在程序中添加实现堆栈警戒功能所需的代码，

但它们仍然依靠底层操作系统来有效地处理堆栈溢出。但是，按照这种方式检测溢出还只是问题的一部分。为了使这类设计更为有效，系统必须能够从堆栈溢出中恢复过来并继续正确地工作。

在一个对安全或任务要求严格的应用中，系统运行时在测试或检测堆栈溢出期间监视堆栈的深度可能并不是一项足够的风险控制措施。对于一些应用，必须确保系统绝对不会越出所分配的堆栈范围；只有通过完整的堆栈深度分析才能证明这一点。这意味着，如果整个程序在同一内存空间运行，则必须对所有代码执行这项分析。不过，如果使用 MMU，分析常可简化。在设计系统时，可将所有关键代码置于一个或多个独立线程内，而这些线程分别在各自的保护内存段中运行。这样，只要对这些关键线程进行堆栈使用分析就可以了。当然，这项简化设计假定当非关键线程溢出其堆栈并失效时，关键线程仍可正确执行。

由于分析工作所需的堆栈使用数据来自汇编语言清单，因此修改代码时，相应模块的堆栈使用信息必须予以更新。如果使用不同的编译器版本，或者改变了优化设置，也必须复核整个分析过程。在理想情况下，编译器将提供每个函数(如果不是每个线程的话)的堆栈使用数量，因为它拥有计算需要的所有信息。例如，瑞萨公司提供有 Call Walker，这是该公司高性能的 Embedded Workshop 开发环境的一部分。这个工具可以图形化地显示每个函数使用的调用树和堆栈，包括运行时库和 C 库的函数。Call Walker 也能找出使用堆栈数量最大的路径。使用这样的工具可以实现步骤 1 到步骤 3 的自动化。