μC/OS-II 软件定时器的分析与测试

引 言

μC/OS-II 操作系统是建立在微内核基础上的实时操作系统，抢占式多任务、微内核、移植性好等特点，使其在诸多领域都有较好的应用。

在 μC/OS-II 2.83 及其以后的版本中，一个较大的变化就是增加了对软件定时器的支持。这使得 μC/OS 实时操作系统的功能更加完善，在其上的应用程序开发与移植也更加方便。在实时操作系统中一个好的软件定时器实现要求有较高的精度、较小的处理器开销，且占用较少的存储器资源。本文在对 μC/OS-II 定时器算法分析的基础上，对定时精度和处理器占用情况进行了分析与测试，其结果在实时系统的设计与应用中具有借鉴意义。

1 定时器实现架构

在 μC/OS-II 操作系统内部，任务的延时功能及软件定时器功能，都需要底层有一个硬件的计数器支持。硬件计数器以固定的频率递减，计数到 0 时，触发时钟中断。这个特定的周期性的中断称为“时钟节拍”。每当有时钟节拍到来时，系统在保存现场和中断嵌套计数加 1 后都会跳到时钟节拍函数 OSTimTick()中，进行软件计数器加 1 和遍历任务控制块，以判断任务延时是否到时。

μC/OS-II 中并未在 OSTim Tick()中进行定时器到时判断与处理，而是创建了一个高于应用程序中所有其他任务优先级的定时器管理任务 OSTmr_Task()，在这个任务中进行定时器的到时判断和处理。时钟节拍函数通过信号量给这个高优先级任务发信号。这种方法缩短了中断服务程序的执行时间，但也使得定时器到时处理函数的响应受到中断退出时恢复现场和任务切换的影响。软件定时器功能实现代码存放在 tmr.c 文件中，移植时需只需在 os_cfg.h 文件中使能定时器和设定定时器的相关参数。

2 μC/OS-II 的软件定时器算法分析

μC/OS-II 中软件定时器的实现方法是，将定时器按定时时间分组，使得每次时钟节拍到来时只对部分定时器进行比较操作，缩短了每次处理的时间。但这就需要动态地维护一个定时器组。定时器组的维护只是在每次定时器到时时才发生，而且定时器从组中移除和再插入操作不需要排序。这是一种比较高效的算法，减少了维护所需的操作时间。

2．1 定时器管理所需的数据结构

一旦定时器被建立，一个定时器控制块(OS_TMR)就被赋值了。定时器控制块是定时器管理的基本单元，包含定时器的名称、定时时间、在链表中的位置、使用状态、使用方式，以及到时回调函数及其参数等基本信息。

在 μC/OS-II 软件定时器中实现了 3 类链表的维护：

```
OS_TMR          OSTmrTbl[OS_TMR_CFG_MAX];
OS_TMR          *OSTmrFreeList;
OS_TMR_WHEEL    OSTmrWheelTbl[OS_TMR_CFG_WHEEL_
SIZE];
```

OSTmrTbl[OS_TMR_CFG_MAX]：以数组的形式静态分配定时器控制块所需的 RAM 空间，并存储所有已建立的定时器控制块。

OSTmrFreeLiSt：为空闲定时器控制块链表头指针。空闲态的定时器控制块(OS_TMR)中，OSTmrnext 和 OSTmrPrev 两个指针分别指向空闲控制块的前一个和后一个，组织了空闲控制块双向链表。建立定时器时，从这个链表中搜索空闲定时器控制块。

OSTmrWheelTbl[OS_TMR_CFG_WHEEL_SIZE]：该数组的每个元素都是已开启定时器的一个分组，元素中记录了指向该分组中第一个定时器控制块的指针，以及定时器控制块的个数。运行态的定时器控制块(OS_TMR)中，OSTmrnext 和 OSTmrPrev 两个指针同样也组织了所在分组中定时器控制块的双向链表。定时器管理所需的数据结构示意图如图 1 所示。

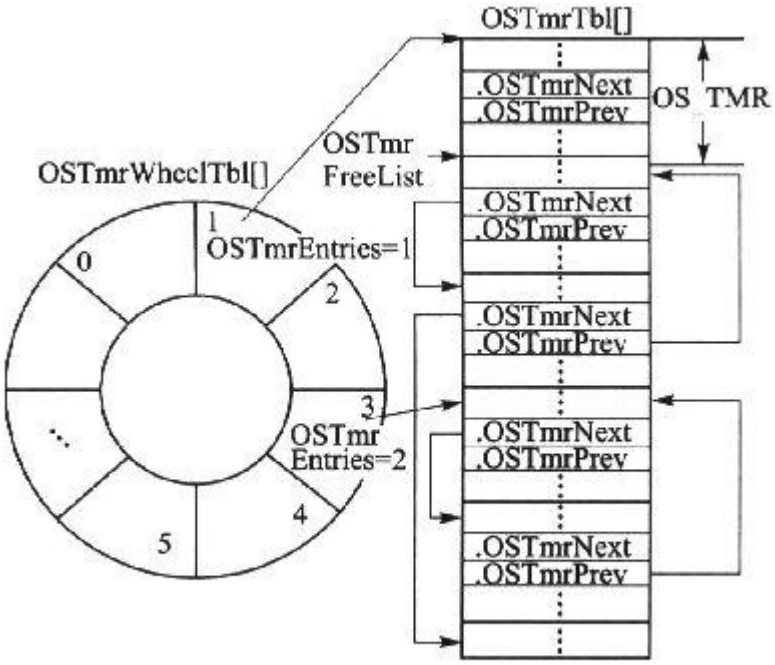

图 1　定时器管理所需数据结构示意图

2．2 软件定时器实现原理

宏 OS_TMR_CFG_WHEEL_SIZE 定义了 OSTmr-WheelTbl[]数组的大小，同时这个值也是定时器分组的依据。按照定时器到时值与 OS_TMR_CFG_WHEEL_SIZE 相除的余数进行分组：不同余数的定时器放在不同分组中；相同余数的定时器处在同一组中，由双向链表连接。这样，余数值为 0～OS_TMR_CFG_WHEEL_SIZE-1 的不同定时器控制块，正好分别对应了数组元素 OSTmr-WheelTbl[0]～OSTmrWheelTbl[OS_TMR_CFGWHEEL_SIZE-1]的不同分组。每次时钟节拍到来时，时钟数 OSTmrTime 值加 1，然后也进行求余操作，只有余数相同的那组定时器才有可能到时，所以只对该组定时器进行判断。这种方法比循环判断所有定时器更高效。随着时钟数的累加，处理的分组也由 0～OS_TMR_CFG_WHE EL_SIZE-1 循环。

信号量唤醒定时器管理任务，计算出当前所要处理的分组后，程序遍历该分组中的所有控制块，将当前 OSTmr-Time 值与定时器控制块中的到时值相比较。若相等(即到时)，则调用该定时器到时回调函数；若不相等，则判断该组中下一个定时器控制块。如此操作，直到该分组链表的结尾。定时器管理任务的流程如图 2 所示。OS_TMR_CFG_WHEEL_SIZE 的

取值推荐为 2 的 N 次方，以便采用移位操作计算余数，缩短处理时间。

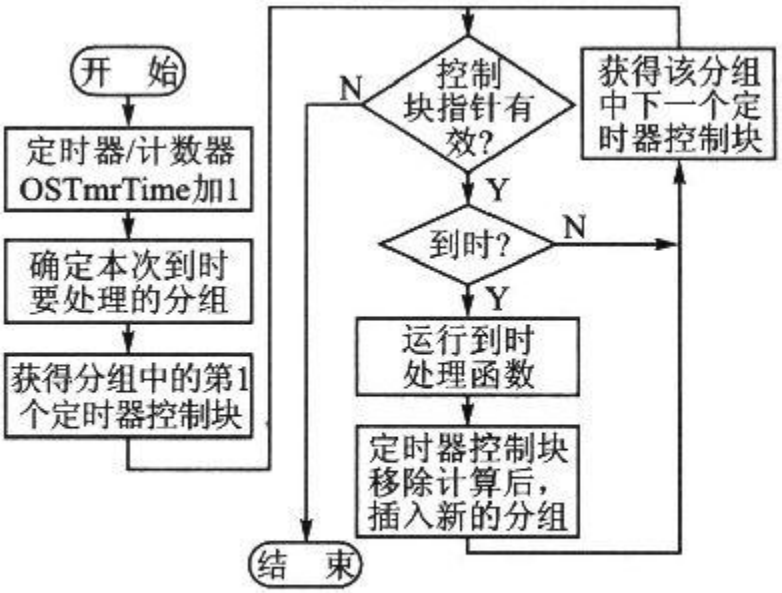

图 2　定时器管理任务流程

2．3 定时器移除和插入操作

定时器的到时处理函数返回后，都要进行该定时器控制块在链表中的移除和再插入操作。插入前需要重新计算定时器下次到时时所处的分组。计算公式如下：

定时器下次到时的 OSTmrTime 值=定时器定时值+当前 OSTmrTime 值

新的分组=定时器下次到时的 OSTmrTime 值％OS_TMR_CFG_WHEEL_SIZE

3 定时器精度与抖动

在 μC/OS-II 操作系统中，与定时相关的功能均基于系统的时钟节拍。系统每秒的时钟节拍数决定了这个系统能分辨的最小时间，定时值只能为最小时间的倍数。每秒的时钟节拍数由 os_cfg．h．文件中的宏 OS_TICKS_PER_SEC 定义。对于不同的应用，该时钟节拍一般在 10～100 次/s 的范围内选取。其对应的时钟中断的时间间隔为 100～10 ms，即时间的最小分辨单位为 10 ms。处理器处理能力越高，每秒的时钟节拍数也相应地越大。下面的数据测试中使用了 ARM9 处理器。最低运行频率为 250 Hz 时，时钟节拍设为 200 次／s，可进行正常的多任务调度。该测试环境下，最小分辨时间为 5 ms。

抖动是指定时器回调函数开始执行的时间与规定的时间相比，或提前或推后的现象。在定时器中抖动总是存在的。下面主要分析 2 种抖动情况及其对定时精度的影响。第 1 种抖动情况如图 3 所示。

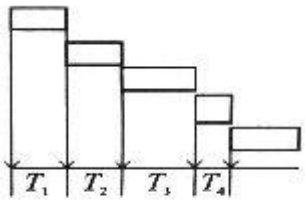

图 3　第 1 种抖动情况

T1：CPU 响应时钟中断，搜索中断号，保存中断现场并跳到时钟中断处理程序 OSTimTick 的时间。

T2：OSTimTick()函数的执行时间。该函数中对任务延时是否到期进行了判断。

T3：恢复现场，退出中断并进行任务上下文切换的时间。

T4：定时器管理任务 OSTmr_Task()判断定时器是否到时的时间。

T4 之后：定时器到时回调函数开始执行。

在 CPU 运行频率固定的情况下 T1 和 T3 的执行时间是相对固定的，而且一般为汇编语言实现，运行速度也较快。而 T2 和 T4 的时间会随着任务和软件定时器数量的增多有所增加。测试时定时器定时值随机选取。表 1 中的时间为不同处理器频率下，定时器个数为 1 和 5 时观测到的最大值。其中，T=T1+T2+T3+T4。

表 1　时钟中断触发到定时回调函数开始执行的时间

处理器频率/MHz	T/μs(1 个定时器时)	T/μs(5 个定时器时)
250	2.976	3.84
300	2.480	3.19
400	1.840	2.41

由测试数据可知，若有定时器到时，则经过几个 μs 的延时，定时器回调函数就可以执行。与 ms 级的定时最小分辨时间相比，其抖动对到时准确性的影响是非常小的。随着处理

器频率的提升和处理能力的加强，这个抖动时间还可以进一步缩小。第 2 种抖动情况如图 4 所示。

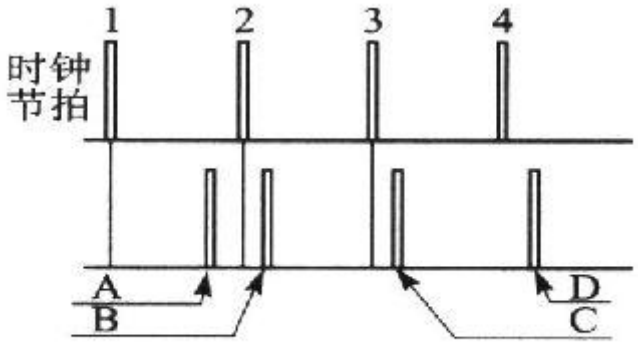

图 4　第 2 种抖动情况

当下一个时钟节拍快到来时(如图 4 中 A 时刻)，一个定时器被开启了。因为系统能分辨的最小时间即时钟节拍的间隔时间，所以定时器开启时记录的当前时钟节拍数为 1，假设定时器的定时值就是一个时钟节拍间隔。定时器开启后，第 2 个时钟节拍马上就到了，由情况 1 的分析可知，这个定时器的回调函数很快就会被执行。B 时刻表示这个定时器回调函数第一次执行。C、D 时刻为回调函数的第 2 次与第 3 次执行时刻，这两次到时是比较准确的。

因此，定时器从开启到回调函数开始执行的时间与定时值相比是有较大抖动的。若此时的时钟节拍间隔为 5 ms，则这个抖动的误差最大即 5 ms。这种情况下的抖动只能通过提高每秒的时钟节拍数来减小。因此，在数据采集或其他定时器的应用中，要留意定时器开启后第一次到时的处理，以免进行错误的判断。

4 定时器管理任务对操作系统性能的影响

定时器管理任务的运行使应用程序可建立的任务数减 1。该任务要求是系统中的最高优先级，因此每次有时钟中断发生它都要运行，必然会消耗掉一定的 CPU 资源。软件定时器功能所占用的 CPU 运行时间(设为 M)的计数公式为：

M=定时器管理任务判断定时器是否到时的时间+定时器回调函数执行时间+定时器控

制块移除与再插入时间+切换到次低优先级任务的时间

首先，采用 μC/OS-II 中自带的统计任务测试定时器管理任务对 CPU 资源的占用情况。测试时系统中开启 3 个任务：定时器管理任务、统计任务和 Idle 任务。定时器到时回调函数在定时器管理任务中执行，增加了管理任务运行时间。为仅测试系统定时器管理功能的开销，测试时将定时回调函数置为空操作。因统计任务要求必须在初始化时建立第一个也是唯一的一个任务中调用统计任务初始化函数 OSStatInit()，所以需在定时器管理任务中进行必要改动，以保证在统计任务初始化后才开启定时器的管理功能。测试用定时器在 main 函数中静态创建。

测试后发现，当定时器数由 1 增加到 8 时，CPU 的使用率一直是 1%。这说明系统开启定时器管理功能后，增加的负荷很小。另一方面因为统计任务计算 CPU 使用率时是整数相除、余数舍弃，这就使得 0～(OSIidleCtrMax/100-1)的计数是不能被分辨的，因此统计任务不能准确地反应定时器管理任务对 CPU 的使用情况。

为了得到更准确的测量结果，采用“处理器占用率”这一指标对管理任务开销进行测试。

处理器占用率=软件定时器功能所占用的 CPU 运行时间(M)/CPU 运行时间

当某一时钟节拍到来时，假设这一时钟节拍下无定时器到时，即“到时定时器回调函数执行时间”和“定时器控制块移除与再插入时间”为 0。若当前系统中所创建的定时器正好在同一个分组中，则管理任务须对每个定时器的到时值进行比较，此时为无定时器到时时定时器管理任务对 CPU 的最高占用率。测试中，将定时器的到时值设定为 8 的倍数加 1，在 main() 函数中静态创建。使所有定时器控制块在同一分组中，当 OSTime 也是 8 的倍数加 1 时，就形成了无定时器到时时定时器管理任务的最坏运行情况。

测试环境为：ARM9(400 MHz)，定时器最小分辨时间为 5 ms(该值作为计算时的“CPU 运行时间”)，硬件计时器的最小分辨时间为 0．02μs。读硬件计时器计算时间，测试结果如表 2 所列。

表 2　定时器管理任务处理器占用率测试结果

定时器数/个	管理任务运行时间/μs	任务切换时间/μs	处理器占用率/(%)
0	0.6	0.82	0.176
2	0.14	0.82	0.192
4	0.20	0.82	0.204
6	0.26	0.82	0.216
8	0.34	0.82	0.232

由测试数据可知建立定时器数为 8 时，处理器占用率远小于 1%。每增加一个定时器，定时器管理任务的运行时间约增加 6．5μs，处理器的占用率约增加 0．01%，任务的切换时间不变。这是在构建的极端情况下测试的，实际应用中的时间要小于这个值。

“定时器控制块移除与再插入时间”在 400 MHz 下测试为 0．22μs。某一时钟节拍处理时，若有定时器到时，则最坏情况是这个到时的定时器控制块在链表的最后，只有执行完整个链表的判断后才开始执行该到时定时器的回调函数。假设定时器数为 8，且链表中最后一个定时器先到时，则该定时器到时的这个钟节拍下处理器利用率为：(0．34μs+0．22μs+0．82μs+定时器回调函数执行时间)/5 ms=(1．38μs+定时器回调函数执行时间)/5 ms。

结 论

本文对 μC/OS-II 软件定时器的算法及抖动问题进行了分析，并在特定平台上进行了测试，较好地反映出了该定时器的精度和对处理器资源的占用情况。测试结果可信度高，为在 μC/OS-II 下的软件定时器应用开发提供了可靠的依据。