|  |  |
| --- | --- |
| 编号： |  |
| 密级： | 内部 |

基于80x51虚拟单片机的计算机数字仿真系统

时间管理模块设计

中国科学院卫星软件评测中心

2015年12月1

目 录

[第一章 概述 1](#_Toc437254199)

[1.1 应用背景 1](#_Toc437254200)

[1.2 设计简介 2](#_Toc437254201)

[第二章 时间管理系统 4](#_Toc437254202)

[2.1 整体架构 4](#_Toc437254203)

[2.2 STW模块 4](#_Toc437254204)

[2.3 DQC模块 7](#_Toc437254205)

[2.4 UTM封装模块 9](#_Toc437254206)

[第三章 性能测试 11](#_Toc437254207)

[3.1 空载测试 11](#_Toc437254208)

[3.2 负载测试 11](#_Toc437254209)

[第四章 用户应用接口 13](#_Toc437254210)

[4.1 接口设计说明 13](#_Toc437254211)

[4.2 使用demo 15](#_Toc437254212)

# 第一章 概述

## 1.1 应用背景

以指令集模拟器（Instruction Set Simulator）为核心的全系统仿真平台有着诸多好处。从系统硬件设计的角度来看，传统的以硬件描述语言(比如Verilog HDL，VHDL等)搭建的平台，由于建模精确到了门一级电路，所以在这样的平台上进行功能验证和架构探索，将需要花费大量的开发时间。同时，这种底层平台对架构设计人员的容错性是非常低的，一旦架构设计出了问题，那么前端到后端的所有中间环节都得修改，甚至推倒重来。相反，基于高级建模语言的全系统仿真平台能够在很大程度上避免上述问题。在设计初期，设计人员利用高级语言快速搭建系统的功能模型，然后根据需要确定的参数属性对该功能模型进行有控制的细化，直到系统所有的参数被最终确定。在这个过程中，设计人员可探索不同的软硬件划分方案，同时对用户需求的功能进行充分的验证。一旦软硬件划分确定后，不同的开发小组就可以同时进行软硬件的开发，并以该全系统仿真平台为最初的测试激励产生平台。

从系统软件设计及测试的角度来看，全系统仿真模型在实际硬件实现之前，为软件的设计和测试提供了一个平台。软件设计人员可在该平台上探索不同的系统软件实现方案，以获得最优策略。对于软件测试，与直接在实际的嵌入式硬件平台上进行测试相比，仿真平台一方面可以大幅降低测试过程中对实际硬件平台的需求，进而降低软件测试的成本；另一方面，也增加了软件测试过程中的可控性和可观测性，仿真平台可提供各种组件以便于软件的测试。最重要的是，仿真平台可加速软件的测试，仿真平台的指令执行速度一般都远大于实际的嵌入式硬件平台的指令执行速度。

基于上述优点，系统仿真平台在国内外得到了广泛的研究与开发，在ISS建模优化领域已诞生了许多重要的成果。然而，在系统仿真平台中的另一个核心概念，时间系统，得到的关注却不多。目前，在仿真平台中对时间的处理主要有如下三种方式：

1）、精确到指令。ISS每执行完一条指令，就将当前仿真时间加一。这种方式的问题在于，不同指令执行所需的时钟周期是不同的，外围设备可能在任意时钟周期发起中断请求。这种时间的不连续流动，可能造成系统丢失外围设备的请求。同时，时间系统受控于ISS的执行，一旦ISS停止执行，时间也就停滞不前了，这与实际硬件情况是不同的；

2）、精确到时钟周期。有两种模式，一种是ISS每执行完一条指令，就将当前仿真时间加上对应指令执行所需的时钟周期数，这种模式的问题同第一种方式一致。另外一种模式是ISS建模以流水线单元为单位，仿真时间加一，就执行对应一个时钟周期内的操作。这种模式的问题在于，时间系统也受控于ISS的执行，无法为外围设备提供时钟；

3）、任意精度。时间系统独立，是一个事件触发系统。所有等待执行的操作均被注册到该时间系统中，时间系统在调度执行了当前仿真时间需要执行的所有操作后，将仿真时间推进到下一个有等待执行事件的时间点上。这种方式多用于硬件描述语言仿真内核中，其问题在于仿真效率很低。

鉴于上述时间系统存在的问题，在本文中将提出一种分层的统一时间管理系统(Universal Time Management)，它独立于ISS，可设置任意精度，支持多处理器系统，同时能够为外围设备提供时钟。该时钟系统便于管理，且运行速度极快，能够满足系统仿真平台对时间系统的绝大部分要求。

## 1.2 设计简介

UTM系统在仿真平台中的位置如图1-1所示：



图1-1 系统仿真平台框架

UTM系统主要用于为系统提供执行节拍，驱动着仿真平台中的ISS以及I/O系统的运行。通常，时间系统将在其内部维护两个事件链表，分别为单次执行事件链表和循环执行事件链表，同时给外部模块提供注册这两种事件的接口。仿真控制模块通过调用时间系统提供的执行接口，步进时间，同时处理队列中在当前仿真时间点到期的等待事件。其核心结构如图1-2所示。UTM系统沿用了这种结构，但其不同之处在于，它采用了分层注册和管理事件的方式。在嵌入式系统中，处理器和外围I/O的执行频率是不同的。处理器中指令执行是在MHz级别，而外围I/O的操作，比如串口的发送和接收，定时器的中断请求等，一般都在KHz级别。如果对这两种事件等同对待，那么时间系统就会浪费大量的资源在无用的外围I/O事件的检测上，严重降低指令的执行速率。



图1-2 仿真时间系统的核心结构

UTM系统将指令执行等高频事件和外围I/O等低频事件进行区分，采用高效的数据结构和算法，以两种不同的方式进行管理。对低频事件，将其注册到一个桶形数据结构中，每个执行点对应一个循环链表，链表中的每个单元对应一个外围I/O事件。执行时只需遍历对应的链表并执行链表单元中对应的回调函数即可。对高频事件，将其注册到一个双通道的队列中，在每个仿真时间点，遍历队列有效通道中的元素，并执行对应的回调操作。低频事件的执行由高频事件管理模块定期驱动。这种分层处理的方式，极大地提高了系统的执行效率。

# 第二章 时间管理系统

## 2.1 整体架构

UTM系统的整体结构大致如图2-1所示：

图2-1 UTM系统整体结构

由图可以看出，UTM系统主要由三部分构成：用于管理高频事件的DQC(Dual Queue Clock)模块，用于管理低频事件的STW（Software Timer）模块以及顶层用于封装的UTM模块。DQC和STW模块均是一个完整的时间管理系统。它们拥有自己的数据对象，并提供对应的操作函数。UTM模块向外提供应用程序接口，其中包括时间系统的创建、删除，高频和低频事件的注册、卸载以及运行控制等操作。 这些接口的实现是通过调用底层DQC和STW模块提供的操作函数来完成的。在系统初始化或运行的过程中，通过调用UTM系统提供的接口，就可以将事件注册到系统中，然后在指定的延迟之后被调度运行。

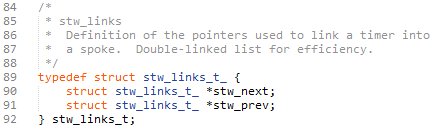
## 2.2 STW模块

STW模块是一个完整的时间管理系统。它采用“桶形”数据结构来管理事件对象，这种数据结构组织如图2-2所示：

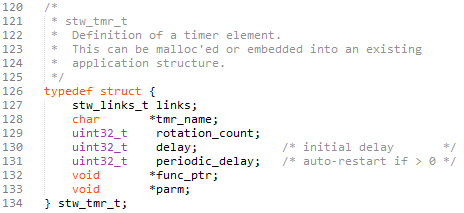


图2-2 STW模块的核心数据结构

这种数据结构由一个数组和对应于每一个数组元素的双向循环链表组成，这里桶的大小即为数组中元素的个数。数组元素对应的数据结构定义如下：



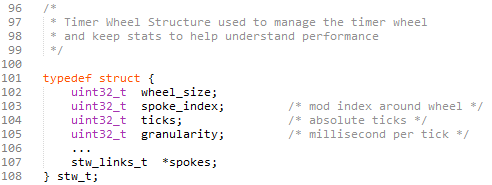
每个数组元素中包含两个指向相同类型的指针，分别用来指向循环链表中的左右两个节点。每个数组元素可看作是双向循环链表的头节点，它标志着一个仿真时间点，也称为一个tick。在该仿真时间点需要执行的事件均被放入循环链表中。循环链表中的每个节点均表示一个等待执行的事件，其数据结构定义如下：



其中links对象用于连接循环链表中相邻的左右两个节点；变量delay表示当该事件第一次被执行时需要延迟的tick数，也就是当该事件被注册后，仿真时间需要前进delay变量所表示的tick数后，该事件才被调度执行。如果该事件需要周期性地被执行，那么执行周期就存放到periodic\_delay变量中。func\_ptr变量中存放回调函数的地址，一旦事件被触发，就执行该回调函数。param变量中存放着回调函数的参数地址。

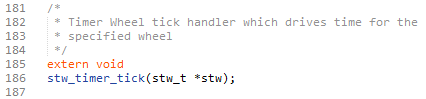
在图2-2中，spoke\_index为数组的索引，表示当前仿真时间点对应的链表头节点在数组中的位置。仿真时间每向前推进一个tick，就将spoke\_index变量的值加一，以指向数组中下一个元素，也就是当前仿真时间点对应双向循环链表的头节点。当spoke\_index增加到数组索引的边界时，就自动回环到数组中第一个元素的索引，如此循环往复，仿真时间不断推进，链表中等待的事件都能得到执行。

如果被注册的事件，其延迟的tick数超出了桶的大小，那么就无法直接在数组中找到一个合适的头节点对应的链表，用来存放该事件。在事件数据对象中的rotation\_count变量可以用来解决这一问题。该变量的值表示spoke\_index需要围着“桶”轮转的圈数，每增加1就表示需要多等待桶的大小对应的tick数。如果事件对象中该变量的值不为0，那么在遍历时就将其值递减一，直到其值为0时，该事件才会被触发。如果当前仿真时间点spoke\_index为3，桶的大小wheel\_size为32，在当前时间需要注册的事件的延迟delay为53个tick，那么该事件的rotation\_count的值为（delay / wheel\_size）1，其添加到的链表的头结点在数组中的索引值为（（spoke\_index + delay % wheel\_size + 1）/ wheel\_size）25。这里默认STW系统的时间精度为1。



数据类型stw\_t用来存放STW的核心数据对象和配置参数。其中wheel\_size为桶的大小，如果该值设置的太小，那么注册到每个事件链表中的节点就会很多，事件对象中rotation\_count的值会比较大，每一次在遍历链表时真正花费在执行事件的回调函数上的时间比较少；如果wheel\_size的值设置的太大，大部分事件链表将为空，浪费存储资源。wheel\_size的值被默认设置为32。变量ticks被用来记录系统仿真时间，以tick为单位。变量granularity为tick的粒度，由于在UTM系统中STW被用来管理外围设置产生的事件，所以这里的granularity以MS为单位，且默认值为1。spokes指针变量中存放上述数组的地址。还有一些辅助变量，在此不再一一详述。

有了上述核心数据结构的分析，在STW系统中注册和卸载定时事件的过程就不难理解了。在此将重点分析STW系统在每个tick的执行过程。其对应的函数声明如下：



在该函数中，首先将仿真时间向前推进一个tick，将spoke\_index变量指向当前仿真时间对应的事件链表，获取该链表的头节点指针。遍历该链表中的每一个元素，如果其rotation\_count域为零，则执行事件对象中注册的回调函数，然后检查其periodic\_delay域的值，如果不为零，则表示这是一个周期执行的事件，则需要将该事件重新注册到对应时间点的事件链表中去；如果值为零，则将其从该链表中删除。反之，如果rotation\_count的值不为零，则只需要将其值递减一个单位即可。当该事件链表中的所有节点都被处理后，那么当前仿真时间点对应的操作就结束了。

## 2.3 DQC模块

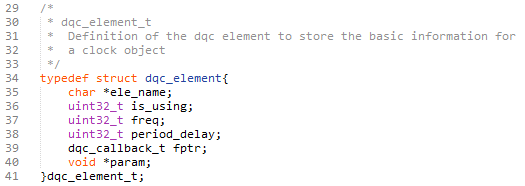
DQC模块也是一个完整的时间管理系统。在UTM系统中用来管理高频事件的注册、卸载以及触发执行等操作。在高频的要求下，为了提高执行效率，减少无效的操作开销，这里将采用静态数组来管理高频事件对象，其结构如图2-3所示：



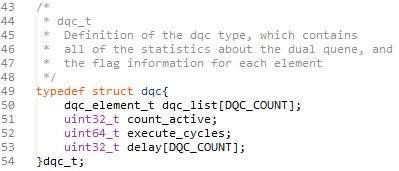
图2-3 DQC系统核心数据结构

这种结构与传统时间系统中采用的数据结构非常类似。系统维护一个固定大小的数组dqc\_list用来存放注册的高频事件。最初，数组为空，在系统初始化或执行的过程中，可以动态地向数组中注册或从数组中删除事件。在每一个时间点处理事件时，并不需要遍历整个数组以寻找被注册的事件。在注册和删除事件时，将维护一个index标记数组。该标记数组中的元素将存贮事件被注册到dqc\_list数组中的位置，也就是下标。变量active\_event的值表示在数组中有多少活跃的事件，也就是被注册的事件。那么在处理过程中，只需要根据index数组中指明的有效位置，从dqc\_list数组中索引对应的事件，直到active\_event变量中值所表示的所有活跃事件均被处理之后结束。

在UTM系统中，高频事件特指处理器系统中的指令执行过程。指令执行在每个时钟周期都会进行，所以，这里所谓的高频，实际上就是指处理器系统的时钟周期。DQC系统可以管理多个高频事件，也就是可以向DQC系统中注册多个指令执行过程，实际上就是多个处理器核心。每个处理器可能在不同的时钟频率下进行操作，那么多个处理器之间怎么进行时钟同步呢？在介绍了DQC涉及的数据对象后，再来详述这个问题。



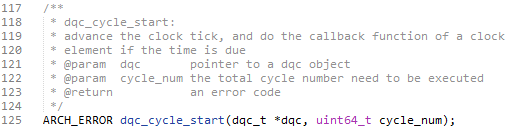
在图2-3中，每个数组元素对应一个事件，其数据结构定义为如上的结构体。其中，变量is\_using表示该数组元素是否已被使用，即是否有事件注册到该元素中；变量freq表示该事件的执行频率，在UTM系统中指处理器的时钟频率；变量period\_delay用来表示该事件每多少个时间单位执行一次。变量fptr为事件对应的回调函数指针，param指针指向其参数地址。



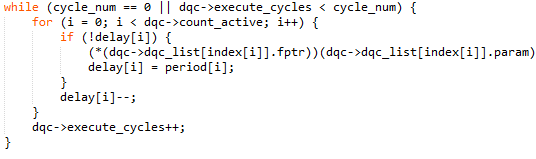
DQC系统对应的核心数据结构定义如上。其中count\_active变量表示注册到系统中的事件数；execute\_cycles表示当前总共执行的周期数；数组delay中存放着每个事件从注册到第一次被执行需要延迟的周期数。

在多处理器系统中，每个处理器的时钟频率不一样，那么注册到DQC系统中的事件执行频率就不一样。那么DQC系统的仿真时间单位怎么确定自己的频率呢？这里采用取所有注册事件频率的最小公倍数作为DQC系统的时钟频率的方式。这样，系统的时钟精度就不小于任何注册进来的事件频率，所有的事件都能在其期望的仿真时间点被调度执行。一旦系统的时钟精度确定，那么就可以确定每个事件每隔多少个系统时钟周期被执行一次，也就是事件对象中period\_delay域中存放的值。比如，CPU1的频率为2MHz，CPU2的频率为3MHz。假设DQC系统中只注册了这两个处理器系统对应的指令执行事件，那么系统的时钟频率为两个CPU频率的最小公倍数，即6MHz。在CPU1对应的事件中，其period\_delay的值为3，即每三个系统周期被调度执行一次；在CPU2对应的事件中，其period\_delay域的值为2，也就是每两个周期被调度执行一次。这样，两个CPU的执行被系统仿真时间交替推进，基本实现了两个处理器系统的同步操作。

下面将详述DQC系统的运行过程。其执行函数的声明如下：



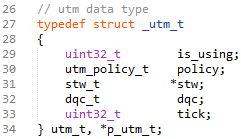
其中，参数dqc指向系统的核心数据结构，cycle\_num表示系统被期望执行的周期数，如果为0，则表示一直执行，直到系统由于其他原因被退出。实现该函数的主体框架如下：



由代码可以看出，该执行函数在每个时钟周期都会去查询所有被注册事件的状态。如果该事件对应的delay值为0，则说明该事件已经到期，需要立即执行，然后用该事件对应的period\_delay域中的（即代码中的period）值来更新delay变量的值。如果事件对应的delay变量的值不为0，或者是已被更新，那么需要将delay变量的值递减一个单位。当所有活跃事件都被处理后，将仿真时间向前推进一个周期，然后进入下一次处理过程。DQC系统的创建、删除，以及向DQC系统中注册和卸载事件的操作比较简单，可直接参考代码。

## 2.4 UTM封装模块

UTM封装模块主要用来组织UTM系统的分层结构，将DQC系统和STW系统进行连接，同时向用户提供方便操作的应用程序接口。其核心数据结构定义如下：



其中is\_using域表示该UTM对象是否被使用，用来支持多个UTM对象同时被创建执行。在UTM系统中，预留了32个UTM对象可被申请使用。枚举类变量policy用来指明DQC子系统在计算系统时钟频率时所采用的机制，有两个值可供选择，MORE\_FAST和MORE\_PRECISE，默认情况下为后者。变量stw和dqc分别指向STW和DQC系统的核心数据对象；tick变量用来存放UTM系统的时钟频率，也即DQC子系统计算出来的时钟频率。

在UTM系统中，DQC子系统用来管理处理器中指令执行等高频率事件，而STW子系统用来管理外围模块产生的中断请求、定时器产生的tick等低频率事件，那么两个系统是怎么连接到一起的呢？在STW子系统中，每个仿真时间点都需要调用执行函数对当前到期的事件进行处理。相比于指令执行，这里的仿真时间点之间的时间间隔要大几个数量级。通常，外设产生的中断事件的间隔在ms级别。在UTM系统的实现中，将STW子系统中的执行函数作为一个事件注册到DQC子系统中，其默认的注册频率为1KHz。那么每过1ms的时间，就会驱动STW子系统执行一次。外部设备产生的延迟事件也以ms为单位注册进STW系统，那么系统时间资源就会如实际硬件系统中那样，绝大部分时间都用在处理器的执行上，对外设的异步事件处理，只需要花费很少的时间。

这里再给出一个例子详细说明。假设在UTM系统中注册了两个处理器系统，其中CPU1的频率为2MHz，CPU2的频率为3MHz，同时在处理器系统中有两个运行在自动装载模式下的定时器，其中断周期分别为5ms和10ms。由于外围设备的执行频率为1KHz，所以可以计算出UTM系统时钟频率为6MHz。CPU1每3个周期执行一次，CPU2每2个周期执行一次，外设事件的处理函数(也即STW系统中每个仿真时间点调用的处理函数)每6000个周期执行一次。在第五次执行时才开始处理第一个定时器事件。

这种将处理器执行和外设异步事件区别对待的方式，避免了时间系统频繁地去查询外设请求而浪费的大量时间，提高了系统的执行效率。UTM系统提供的用户应用程序接口将在第四章详细介绍，在此不再赘述。

# 第三章 性能测试

## 3.1 空载测试

这一章主要给出对UTM系统进行性能测试所得到的结果。该系统运行的宿主机环境如下：宿主机的型号是ThinkServer140，处理器频率3.3GHz，双核四线程，内存大小4G，磁盘大小1T，操作系统为64位的Windows7 ultimate SP1。这里，对UTM系统的测试分为空载测试和负载测试两部分。其中空载是指在注册事件对应的回调函数中，只给出统计信息，而不作任何其他有效的处理。在UTM系统中注册的事件分为处理器事件和外设事件，此处处理器事件以‘#’开始编号，外设事件以‘$’开始编号。测试结果如表3-1所示。

表3-1 UTM系统空载测试结果

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **序号** | **CPU事件执行频率（M/s）** | | | **外设事件执行频率（M/s）** | | | | **系统效率（M/s）** |
| **#1** | **#2** | **#3** | **$1** | **$2** | **$3** | **$4** |
| **1** | 293.26 |  |  |  |  |  |  | 293.26 |
| **2** | 289.32 |  |  | 2.86 |  |  |  | 292.18 |
| **3** | 145.16 | 145.23 |  | 2.81 |  |  |  | 293.20 |
| **4** | 144.83 | 144.25 |  | 2.67 | 2.67 |  |  | 294.42 |
| **5** | 143.25 | 143.71 |  | 2.54 | 2.54 | 2.54 |  | 294.58 |
| **6** | 142.96 | 142.63 |  | 2.49 | 2.49 | 2.49 | 2.49 | 295.55 |
| **7** | 97.43 | 97.41 | 97.52 |  |  |  |  | 292.36 |
| **8** | 96.65 | 96.65 | 96.72 | 2.80 |  |  |  | 292.82 |
| **9** | 96.03 | 96.11 | 96.03 | 2.75 | 2.75 |  |  | 293.67 |
| **10** | 94.46 | 94.41 | 94.46 | 2.63 | 2.63 | 2.63 | 2.63 | 293.85 |

CPU的注册频率为1MHz，外设的注册频率为1KHz，也就是每隔1ms执行一次。由表中的结果可以看出，UTM系统空载时，注册到系统中的事件的执行频率可达到300M/s的速度。同时，随着注册外设事件的增加，处理器的执行速度却几乎保持不变。这一点是与其他时间系统最大的区别。系统中如果存在多个处理器系统，那么将均分UTM系统的执行速率。

## 3.2 负载测试

对UTM的负载测试是在实际的仿真平台中进行。这里采用80c32嵌入式仿真平台来对UTM系统进行测试。80c32嵌入式仿真平台模拟了80c32单片机上的所有资源，包括一个51架构下的处理器，三个定时器，一个串行通信接口和一个中断控制器。其中中断控制器在处理器的内部实现。处理器的时钟频率为12MHz，由于51架构下的处理器在执行指令时是以机器周期为单位的，而一个机器周期等于12个时钟周期，所以这里处理器在UTM系统中注册的时钟频率为1MHz。在处理器执行指令的过程中，会动态地注册定时器事件到UTM系统中。此处将三个定时器均设置为周期计时模式，其计时周期均为64\*1024，也就是在这么多个机器周期后，定时器将溢出，触发定时器中断事件的执行。测试结果如表3-2所示。

表3-2 UTM系统负载测试结果

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **序号** | **CPU（MIPS）** | **外设事件执行频率（M/s）** | | | | **系统效率（M/s）** |
| **T0** | **T1** | **T2** | **UART** |
| **1** | 120.34 | 2.56 | 2.56 | 2.56 |  |  |
| **2** | 122.58 |  |  |  | 0.27 |  |

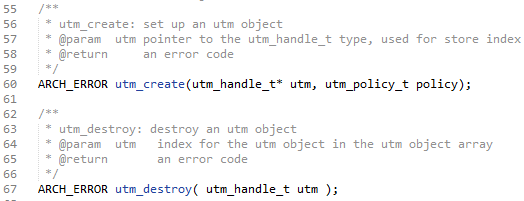
CPU的注册频率为80c32单片机的机器周期对应的频率，为1MHz。在实验1中，三个定时器均被设置为每隔1ms产生计数溢出；在实验2中，定时器2工作在波特率产生器模式，提供给串口的波特率为1.2K左右。由表中的结果可以看出，CPU的指令执行速率为120MIPS左右。相对与UTM空载时下降了一半还多。这是由两个原因引起：首先是，在负载情况下，处理器对应的事件回调函数中将处理指令执行，中断检查等操作；其次是51指令集中大部分指令都是需要两个机器周期才能执行完成，也就是处理器对应的事件回调函数平均需要执行两次，才能执行完一条指令。这个仿真速度已比实际的80c32单片机的指令执行速度快出了两个数量级，能够满足嵌入式仿真平台的大部分需求。且外设也能够如在实际硬件平台中那样正确的执行。

# 第四章 用户应用接口

## 4.1 接口设计说明

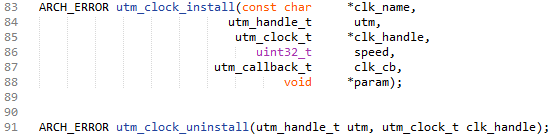
本章主要介绍UTM系统提供给用户的应用程序接口。这些接口简单易用，屏蔽了UTM系统的所有实现细节。接口主要分为五类：UTM的创建与删除、高频事件的注册与卸载、低频事件的注册与卸载、运行控制以及配置管理接口。下面将逐一介绍。

UTM的创建与删除接口函数声明如下。



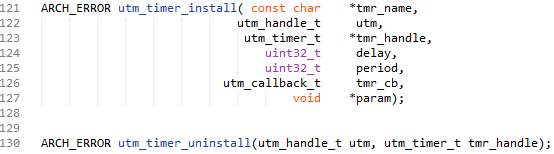
在utm\_create接口函数中，参数utm为handler指针，当UTM对象创建成功时，就将该对象的handler放入utm指针指向的地址中去。UTM对象的handler实际上就是它在预留UTM对象数组中的位置。参数policy指明UTM系统的操作模式。当UTM对象创建成功后，将返回ARCH\_ERROR\_OK。在utm\_destroy接口函数中，传递UTM对象的handler，然后UTM对象将被释放。在内部，就是将其正在使用标志位置零。

高频事件的注册与卸载接口声明如下：



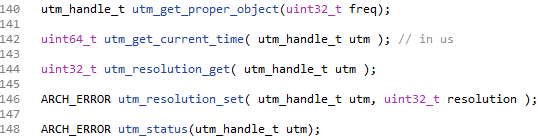
在utm\_clock\_install接口函数中，参数clk\_name为注册事件的名字，utm参数的类型为UTM对象的handler，它指明该事件将被注册到哪一个UTM系统中。由于在整个仿真平台中支持多个UTM同时运行，所以该参数是必要的。clk\_handle指向该高频事件的handler，后续对该事件的操作都是通过这个handler来进行的；speed参数为该事件的执行频率，在绝大多数情况下，高频事件为处理器的指令执行过程，所以这里的事件的执行频率也就是处理器的时钟频率；参数clk\_cb为该事件对应的回调函数，param指针中存放着该回调函数参数的地址。如果事件注册成功，将返回ARCH\_ERROR\_OK。在utm\_clock\_uninstall接口函数中，传递该事件以及其注册所在UTM系统的handler，然后相应的事件对象将被释放。如果操作成功，将返回ARCH\_ERROR\_OK。

低频事件的注册与卸载接口声明如下：



在utm\_timer\_install接口函数中，参数delay和period的单位在UTM系统中均为ms。参数delay表示该事件从注册到第一次执行时的延迟，period表示该事件的执行周期。参数tmr\_cb为回调函数的指针。低频事件被注册到STW子系统中，如果注册成功，将返回ARCH\_ERROR\_OK。在utm\_timer\_uninstall接口函数中，，传递该事件以及其注册所在UTM系统的handler，然后相应的事件对象将被释放。如果操作成功，将返回ARCH\_ERROR\_OK。

配置管理接口声明如下：



当多个UTM系统同时运行时，在进行处理器系统中外部事件的注册过程中，需要确定将该事件注册到哪一个UTM系统。接口utm\_get\_proper\_object用来实现这个功能。它的参数freq表示产生事件的外设所在处理器系统的时钟频率，该接口函数将返回该处理器事件所注册的UTM系统的handler，以用于外设事件的注册。接口utm\_get\_current\_time用于返回参数utm标识的UTM系统的当前仿真时间。接口utm\_resolution\_get和utm\_resolution\_set分别用于获取和设置UTM系统的时间精度，也就是系统的时钟频率。接口utm\_status用于显示UTM系统的当前状态。

运行控制接口声明如下：

## 4.2 使用demo

这里给出一个两个UTM系统以多线程的方式同时运行的示例程序。UTM系统提供的用户接口简单易用。用户首先需要创建UTM对象，然后分别注册处理器和外部事件。在实际中，实际上只用注册处理器的指令执行事件，外设事件的注册是在处理器的执行过程中动态进行的。注册之后，需要将UTM内部的分层结构进行连接，也就是调用utm\_prepare接口函数。这之后，可以调用配置接口对UTM系统的参数进行设置。至此，准备工作就已结束，UTM系统可以开始运行了。由于是在多线程环境中，为了避免主线程过早退出，我们需要等待UTM线程执行结束后返回的通知。示例程序如下文所示。

