论文题目 嵌入式操作系统中的
任务分析机制的设计与实现

专业学位类别 工 程 硕 士

学 号 201291100136

作 者 姓 名 王智广

指 导 教 师 陈绍刚 教 授

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

嵌入式操作系统中的任务分析机制的设计与实现

（题名和副题名）

王智广

（作者姓名）

指导教师 陈绍刚 教 授

电子科技大学 成 都

石 媛 高 工

中国移动北京有限公司 北 京

（姓名、职称、单位名称）

申请学位级别 硕士 专业学位类别 工 程 硕 士

工程领域名称 软 件 工 程

提交论文日期 2013.09 论文答辩日期 2014.1.12

学位授予单位和日期 电子科技大学 2014 年 6 月 27 日

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

DESIGN AND IMPLEMENTATION OF TASK ANALYSIS MECHANISM IN EMBEDDED OPERATING SYSTEM

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Major: **Master of Engineering**

Author: **Wang Zhiguang**

Advisor: **Chen Shaogang**

School : **School Of Mathematical Scienes**

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

作者签名：_____ 日期： 年 月 日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名：_____ 导师签名：_____

日期： 年 月 日

摘　要

在嵌入式系统，特别是实时嵌入式系统中，不仅对逻辑的正确性有着基本的要求，而且对任务的执行时间有着更高的要求。在操作系统中，任务的执行时间会受到任务的调度、资源的竞争、中断等因素的影响。传统的系统级调试无法获取这些信息，而任务级调试可以，它可以提供更高层次的调试功能，能够在操作系统的框架中分析被调试程序的运行行为，从而发现任务级的故障。

本论文主要针对嵌入式操作系统中的任务级分析机制进行研究，研究如何针对任务的执行信息进行实时记录、实时显示、以及快速重放。本文的主要贡献如下：

（1）提出了一套完整的针对嵌入式操作系统执行过程中的任务信息进行分析的机制，包括任务信息的实时记录、实时显示、以及快速重放。实时记录机制可以在操作系统运行过程中实时地将所有任务相关的信息完整记录下来，实时显示机制可以以图形化的方式实时显示出所有任务的当前运行状态，重放机制可以根据用户指定的条件对过去的任务进行快速的重放。

（2）基于 AT697 处理器软件模拟平台和实时操作系统实现了一个完整的任务分析系统。该系统包含一个 AT697 软件模拟器，它完整模拟了 AT697 处理器的指令架构以及结构功能，在执行过程中能够监测操作系统对任务信息的修改，并对其进行记录。系统还包括一个前台的 IDE 界面，该界面可以实时显示操作系统运行过程中的任务信息变化情况，并且可以根据之前的记录针对特定的信息进行重放。

（3）提出了一种针对实时记录信息进行压缩的优化机制，以节省实时记录所需的空间开销。我们根据任务信息记录的不同类型分别采用不同的方法进行压缩。实验结果表明，采用压缩机制后的任务分析系统相比压缩前最大可以减少 70%的记录大小，而系统整体执行速度仅下降不到 10%。

关键词：嵌入式操作系统，任务级调试，实时记录，实时显示

ABSTRACT

Embedded systems, especially real-time embedded systems, not only require the logic correctness, but also have strong demand for the execution time. However, the execution time is affected by many factors, such as task scheduling, resource contention, or interrupts. In this situation, traditional system-level debugging mechanism is unable to find this information. It is the superiority of task-level debugging mechanism. Task-level debugging can provide much higher debugging function and analyze the behavior of debugged programs in the framework of operating system, making it suitable to find the task-level bugs.

This thesis gives a systematic research on the task analysis mechanism in embedded operating systems, focusing on real-time recording, real-time display, and fast replay mechanism. The main contributions of this thesis are as follows:

1. We provide a mechanism to analyze the dynamic task information of embedded operating systems. It includes real-time recording, real-time display, and fast replay. Real-time recording can record all the task information on the fly. Real-time display can draw the states of all the executing tasks. Replay mechanism can show the past task information according to different conditions.

2. We implement a task analysis system based on AT697 software simulator and real-time operating system. This system includes an AT697 simulator which simulates the architecture and function of AT697 processor. The simulator can monitor the modification of task information and make a record for that. The system also has an integrated development environment (IDE) which can display the states of task execution on the fly. The IDE is also responsible for the reply of task information.

3. We introduce a compression mechanism to reduce the size of recording file. We use different comparison methods for different kinds of task records. Experiment results show that we can reduce almost 70% of the file size, while the slow-down ratio is less than 10%.

Keywords: embedded operating system, task-level debug, real-time recording, real-time display

目 录

第一章 绪 论

从功能层面来划分，调试器可以分为两种，系统级调试器和任务级调试器，这其中，前者又称为源代码级调试，而后者又称为操作系统级调试。源码级调试是我们使用的比较多的，常用于调试目标程序中的一些功能实现错误，它可以通过解析带调试信息的目标二进制文件，将针对二进制文件的调试转化为源代码级的调试信息，以方便用户使用，用户可以进行一些常规的调试操作，如断点设置，单步执行，打印变量等等。源代码调试通常针对的是一个函数或者一串指令序列这样的，一般不会考虑到操作系统中多任务调度的影响，而如果一些问题是由于操作系统中任务调度相关机制引起的话，那么使用源码级调试将难以发现此类错误。

针对该问题，任务级调试机制被引入进来，任务级调试针对的是整个操作系统以及操作系统之上的应用程序，除了传统的源代码级调试所包含的调试功能之外，任务级调试还能识别出操作系统任务调度相关的信息，能够监控到任务的映射关系以及中断产生等其他系统状态。从操作系统的角度来看，用户所调试的目标是一个独立的任务或者进程，其行为是受到操作系统中其他任务的影响的。

在实时嵌入式系统中，我们往往不只是要求保证应用程序的逻辑正确性，同时对操作系统的任务调度有着非常高的时序要求。在嵌入式实时操作系统中，任务的调度又往往受到各种因素的影响，比如，资源的竞争，外部的中断等等，具有很大的不确定性，而任务调度的异常又可能引起整个实时系统的失效，因此，我们希望在实时嵌入式系统中，不光提供源码级的调试机制，还应提供更高层次的、系统级的调试机制，使得用户能够在操作系统的层面分析目标程序的行为，发现任务级的故障问题。

1.1 国内外发展现状

1.1.1 嵌入式系统调试技术

在传统的嵌入式系统中，早先是由程序员手工检查程序源代码，查找其中的错误，如果发现有错误就修改，将修改好的程序生成目标代码烧录到 EPROM 里面，再运行系统，如果发现新的错误，再重新查找源代码 bug，循环上述过程。可以看到，这种方式非常耗时，调试起来也很不直观，很不方便，用户也难以定位到真正的错误到底在哪里[1]。后来，人们开始使用发光二极管或者示波器/逻辑分析仪这样的仪器来进行故障检测，还有的方式是通过串口来输出一些调试信息，

供用户进行故障的定位。

此后，又陆续出现了 ROM 监控器（ROM Monitor）和内部电路仿真器（In-Circuit Emulator, ICE）这样的调试机制。ROM 监控器是一小段驻留在 ROM 中的程序，它可以运行在 host 机上的调试操作软件进行交互和通信，可以通过串口或者网络的方式进行。它的硬件结构比较简单，只需要少量的内存，加上一个用于通信的端口基本上就可以了。ROM 监控器可以完成诸如代码下载、断点设置、单步执行、查看修改内存或者寄存器等常规的调试功能。ROM 监控器这种调试方式成本较低，但是也有一些不足，比如无法直接调试从 ROM 里运行的程序，而且监控器代码本身也需要占用 ROM 空间，另外，如何对 ROM 监控器本身进行调试也是个问题[2-7]。再往后，ICE 是用来仿真 CPU 的核心，他可以实时的检测 CPU 的工作状态，同时不对运算器的正常工作造成影响。它能够支持普通调试软件所支持的功能，比如断点设置、实时跟踪、性能和端口分析等。当同 host 机上强大的调试软件一起使用的时候，ICE 能够实现相对比较全面的调试功能[8]。不过，ICE 机制同样也有一些不足之处，比如，它比较昂贵，而且一般需要一个比较特殊的 CPU（外合 CPU），在实际环境中，并不是所有的 CPU 都可以作为外合 CPU。

在线调试(On-Chip Debugging, OCD)是新发展的调试技术，OCD 的实现方式通常有两种，一是把用于监控的代码嵌入到 CPU 中，另一种是在 CPU 中添加一些额外的硬件逻辑。OCD 端口和 host 机的前端调试软件可以通过适配器的方式相连。在原理上来看，OCD 和前面提到的 ROM 监控器是比较类似的，但是 OCD 不需要专门的通信端口以及专门的程序[9]。目前，摩托罗拉的背景调试监测(Background Debug Monitor，BDM) [1,10]和 JTAG(Joint Test Action Group) [11,12]是两种使用的比较多的 OCD 接口。后来，还有一些增强型的 OCD 技术，使用相关的技术可以用增强型 OCD 来对 ICE 调试机制进行模拟，以保证低成本的同时能够提供更加强大一些的调试功能。

上面介绍了嵌入式系统中的一些调试机制的演化，需要注意的是，相比于普通 PC 平台上的软件调试，嵌入式系统给的软件调试有一些需要特别考虑的地方，比如，如何实现调试器和被调试端的通信，调试器如何访问和控制被调试端的程序，调试器如何识别出被调试端操作系统中的任务相关信息，调试器如何支持不同的目标调试硬件结构等等。针对上述问题，现在主要的解决方法包括以下几类：

1. 片上调试方案

这种调试方法是一种硬件支持的调试方案，它通常在处理器内部加入一些特殊的硬件结构，它会检测处理器的执行过程，当处理器执行到某个特殊阶段，触发了设定的调试条件时，处理器就会暂停执行，此时调试器可以利用和被调试端

的通信接口访问到被调试端的内部资源，比如存储器、寄存器等。需注意的是，这个额外的硬件结构可以是一个纯硬件的模块，也可以是一个基于指令微码序列的控制模块，同时也还包括用户接口[13]。片上调试方案的优点在于性能较好，不需要软件实现太多的功能。但是纯硬件结构支持的方法灵活性不是太好，调试机制难以修改，而通常基于微码序列的方式则灵活性更好一些，可以方便更新和替换。

2. 调试代理方案

调试代理方案[14]是一种软件的调试方式，他需要在被调试目标机上安装一个调试代理软件，这个调试代理软件支持一些基本的调试功能，然后调试器与这个调试代理之间进行通信，通过调试代理实现对目标软件的调试。

在这种机制下，首先需要先建立调试器和目标机上的调试代理之间的通信连接，然后调试代理程序等待主机上的调试器发送调试命令。开始调试后，用户通过主机上的调试器发出调试命令，调试器将其生成指令，通过约定的通信协议发送给目标机上的调试代理，调试代理解析该命令，并控制调试程序完成相应的操作，类似地，调试代理程序也可以获取到被调试程序的一些状态信息，并按照通信协议反馈给主机上的调试器，然后调试器解析后反馈给用户。

调试代理软件与调试器的交互通过插入调试端口通信模块来完成，在启动过程中，需要先完成调试代理软件的初始化，尤其是调试通信端口的初始化[1]。

需要注意的是，调试代理机制虽然是个软件方法，但是调试代理软件还是需要占用目标机上的一些硬件资源和通信端口，被调试程序一般是通过调试器装载到目标机上的 RAM 中运行。相比于片上调试方案，调试代理方式的优点在于避免了额外的硬件支持[1,3]。

1.1.2 软件调试工具

对于软件调试机制，目前我们所考虑的已经不只是传统的那些注入断点、单步执行这样的调试过程，而往往还需要其他辅助机制的加入来协助故障调试，具体的机制包括关键性能统计和显示、指令 trace 收集等。一个好的调试工具应该尽可能少的对被调试机器的正常执行产生影响。目前的软件调试工具中，源代码级的调试工具相对比较成熟，也已经有很多成熟的使用工具，比如基本上现有的交叉编译工具都能提供基本的源代码级的调试支持。这种技术国内外已经基本上都掌握了，也有一些相应的软件产品，单纯从技术上来看，国内和国外的水平差距并不大[15]。

不过，在任务级调试机制这块，目前，国外的研究工作相比于国内还是处于

领先的地位。由于嵌入式系统的任务级调试机制往往需要依赖于一个成熟的嵌入式操作系统，因此，即使在国外，支持任务级分析调试的工具也往往是较大的一些嵌入式操作系统的开发商提供的，比如风河公司在 Tornado 环境中提供了对应的调试工具，就是和嵌入式操作系统 VxWorks 绑定在一起的。此外，还有一些其它嵌入式厂商也提供一些类似的调试工具，比如 ARM 公司的 RealView 和 Metrowerks 的 CodeWarrior 等[4] 。但是目前总体而言，风河公司的 Tornado 开发工具在技术上是有优势的，其配套的调试工具也相对更加完善一些，能够绑定单个任务进行调试[16]。

国内的嵌入式系统开发目前仍处于快速发展过程中，进步很快，也有不少研究成果，比如文献[17,18,21,22,23,24]中都有相关的研究工作介绍。但是，我们也看到，国内在嵌入式系统任务级调试这块的研究工作仍然非常薄弱，目前的研究大多数仍然是基于利用第三方或者一些开源操作系统进行一些应用程序的开发，而缺乏自主完善的实时嵌入式操作系统的研究和开发工作，针对操作系统的任务级分析工具的研究就更少了，大多可能也还停留在学术研究层面[19,20]，这导致目前国内仍很少出现成熟的任务级调试分析工具[15]。目前国内相对比较成熟的嵌入式调试工具的一个代表是科银京成公司的 Lambda 调试工具[25]，但是它仍然缺乏针对操作系统的任务级分析调试机制支持。

1.1.3 发展趋势及意义

从当前嵌入式软件的发展方向的角度来看，目前的嵌入式软件的开发对各类开发工具的依赖越来越强烈，并且，软件本身的复杂性也在不断提升，规模越来越大。实际应用的嵌入式系统往往需要操作系统的支持，并且很可能是多任务实时操作系统。在这种环境下，往往要求能够支持任务级的调试机制，所以支持操作系统中任务级分析调试的工具对于未来嵌入式系统的软件开发是非常重要的。

目前，仍然缺乏有效的进行任务级调试的工具，国外的 Tornado 工具虽然能够提供一些任务调试支持，但是目前也仅支持针对单个任务的绑定，无法实现多任务的调试支持。而在国内，目前还基本上没有太成熟的用于任务级分析调试的软件工具。在实时嵌入式系统中，我们往往不只是要求保证应用程序的逻辑正确性，同时对操作系统的任务调度有着非常高的时序要求[26]。在嵌入式实时操作系统中，任务的调度又往往受到各种因素的影响，比如，资源的竞争，外部的中断等等，具有很大的不确定性，而任务调度的异常又可能引起整个实时系统的失效，因此，我们希望在实时嵌入式系统中，不光提供源码级的调试机制，还应提供更高层次的调试机制，使得用户能够在操作系统的层面分析目标程序的行为，发现任务级

的故障问题，而任务级调试机制就是用于解决这一问题的。

因此，本论文对于任务级分析调试机制的研究，符合未来实时嵌入式系统的发展需要，能够弥补我国在操作系统任务级调试和分析工具方面的不足，有助于推动我国的嵌入式操作系统和应用软件的快速发展。

1.2 本文主要工作

从上面的介绍中，我们可以看到，在传统的、简单的嵌入式系统开发过程中，目标软件调试工作最终需要采用交叉调试方式进行。借助于常规调试工具用户只能通过设置断点等方式控制程序执行，实现基本调试功能。所看到的程序执行现状，也只是目标机当前程序寄存器、内存、符号信息等基本信息。而在具备操作系统支持的嵌入式系统中，我们可以采用多任务编程，程序中的很多问题是由于任务级的切换所引起的，为了便于发现这类问题，我们迫切需要获得操作系统执行过程中的任务级信息，这就要求调试分析器能够提供任务级的特殊调试支持。

本论文拟针对嵌入式操作系统的任务运行状态进行记录和分析，研究成果可以用于相关的任务级调试机制，具体的研究内容包括：

（1）任务信息的实时记录。在任务的调度执行过程中，动态的实时记录下任务的所有相关信息，用于后续的分析调试。

（2）任务信息的实时显示。在任务的调度执行过程中，实时地将任务的相关信息以图形化的方式显示出来。

（3）任务信息的快速重放。根据之前记录的任务运行信息，可以结合指定条件进行快速的任务执行轨迹重放。

（4）任务信息记录格式的压缩优化。通过对记录信息进行压缩，可以有效减少系统所需的记录文件空间大小。

在本文中，我们基于上述机制开发实现了一个图形化的任务级调试分析器，该调试分析器的优点主要表现在：

（1）全面的记录多任务操作系统中的任务调度情况

该调试分析工具能够全面的记录多任务操作系统的任务情况，包括任务块的基本信息，其中包括任务有效标识，任务名，所有任务队列节点，任务当前优先级，任务初始化优先级，任务状态，任务选项，任务函数入口地址，任务参数，任务堆栈大小，任务堆栈基址，轮转剩余时间片，延时时间，睡眠队列中差分延时时间，占有的互斥信号量个数，任务整点上下文，任务浮点上下文，睡眠的队列节点，信号量或消息阻塞队列节点，同优先级任务队列节点，等待的信号量，等待的消息队列，存放消息的内存缓冲区指针，存放消息的最大字节数共 24 个任

务块信息。还包括任务运行起始时间和运行时间的信息。该信息全面的涵盖了在嵌入式多任务 OS 中运行的所有任务的运行情况，可以为图形化的实现提供完整的数据支撑。

（2）直观的将多任务的运行轨迹通过图形化的方法实现

图形化分析工具提供给用户完整并且直观的图形化展示界面。该界面会根据记录的任务数据，生成横轴为运行时间，纵轴为运行状态的图标，并可以清晰的查看每个状态的任务运行信息。该界面直观，简洁，能够为程序的任务级分析提供有力的帮助。

（3）能够协助调试人员快速的定位程序错误或帮助程序员找到优化程序的方法

该图形化分析工具，完整的展示了系统在运行多任务操作系统时的任务运行状态，调试人员可以根据图形界面，清晰的得知任务的调度情况和任务运行的实际状态，从而快速定位程序优先级设计，任务锁设计或多任务抢占设计的缺陷。能够为程序的调试分析提供最强有力的保障。

1.3 实验平台介绍

本文研究的是嵌入式平台上的操作系统任务分析机制，因此，实验平台主要由两个部分构成，一是嵌入式硬件系统平台，二是基于该硬件平台的操作系统及其运行程序，具体来讲，我们将基于 SPARC v8 架构的 AT697 嵌入式硬件平台以及多任务实时操作系统（TOS）进行，不过为了实验的方便，我们没有在真正的 AT697 硬件平台上进行，而是基于一个 AT697 软件模拟平台（Sim697）来开展，基于模拟器上的调试研究工作相对比较方便开展，之前也有一些相关的工作[27,28]，下面我们将简单介绍一下本文工作所使用的软件模拟实验平台。

Sim697 是基于 SPARC V8 指令集的 AT697 芯片的模拟系统，该模拟器完整模拟了 AT697 处理器的架构，其实现方式参考 SimOS-Goodson[29]模拟器开发，主要包括两个部分的工作：一是对 AT697 指令集的模拟，二是对处理器内部结构的模拟。

Sim697 模拟器首先在 ISA 层面实现了对 SPARC V8 体系结构的支持，主要包括两个方面的功能：

- 首先是需要定义目标机器结构

这部分的定义包括目标结构的寄存器定义、指令译码和功能实现等。主要的工作包括实现 SPARC 指令的译码、定义等。

- 其次是完成目标架构下二进制文件的装载

Sim697 模拟器需要对 Linux/SPARC 的可执行文件进行支持，完成操作系统或者应用程序的装载过程。

然后，Sim697 还针对 AT697 处理器的结构进行了完整模拟，包括处理器核以及各种片载设备，比如 UART、TIMER、Watchdog、中断控制器等，因此，Sim697 实现了针对 AT697 硬件平台的全系统模拟，可以直接运行不加修改的操作系统代码，在本文中，我们的模拟平台可以直接运行实验用的嵌入式操作系统 TOS。

本文所做的任务实时记录和实时显示均是基于 Sim697 进行，我们在 Sim697 的基础上，实现了一个图形化的集成开发环境（IDE），可以用于显示 TOS 运行过程中的任务状态。

1.4 本论文的结构安排

本文针对嵌入式操作系统中的任务分析调试机制进行研究，主要针对任务分析机制中的任务信息实时记录、实时显示、以及快速重放机制进行了研究和实现。本文主要包括如下内容：

第 1 章阐述了当前国内外在任务级分析调试方面的研究现状以及趋势，介绍了本文的主要研究工作，以及本文的实验平台。

第 2 章介绍任务分析机制的具体设计。从 TOS 实时操作系统的任务调度原理开始分析，依次介绍了实时任务信息的实时记录、任务信息的实时显示、以及任务信息的快速重放机制设计。

第 3 章详细详细介绍了基于 AT697 处理器模拟器以及 TOS 实时嵌入式操作系统实现的支持任务分析机制的原型系统，介绍了系统中实时记录、实时显示、重放机制的实现，并给出了一些优化措施。

第 4 章基于第 3 章给出的软件系统实现，针对任务分析机制进行了详细的实验评估，并对结果进行分析。

第 5 章对本文所做工作进行了总结，并提出了对未来研究的展望，即今后工作的主要目标和研究方向。

第二章 任务分析机制的设计

在任务分析机制中，主要包括任务信息的实时记录、任务信息实时显示、以及任务信息的重放三大功能。在本章中，我们先介绍一下 TOS 操作系统的详细实现机制，然后进一步介绍一下和任务分析机制相关的这三大功能的实现机制和涉及到的一些关键技术。

2.1 TOS 操作系统

2.1.1 TOS 操作系统的主要特点

TOS 是面向于航天应用的 AT697 处理器开发的，一种可移植的、抢占式的、可裁剪的实时多任务操作系统内核。其具有如下特点：

- 很小的操作系统内核

TOS 只是一个实时操作系统内核，仅仅包含了任务管理，时间管理，任务调度，任务间的通信和同步和内存管理等基本功能，其他功能可以通过扩展的方式进行支持，但并不包含在操作系统内核实现当中。

- 便于移植

TOS 的代码中除了极少部分与处理器密切相关的代码是用汇编语言编写的以外，其大部分代码都是使用 C 语言编写的。而且 TOS 的代码将总量约为 200 行的汇编语言部分压缩到了最低限度。因此，TOS 可以很容易移植到各类嵌入式处理器上，需要修改的主要是少部分的嵌入式汇编代码实现。

- 支持多任务运行

TOS 中可以支持任意多个任务的创建，从而适应不同规模的调度场景。并且，用户还可以手动更改操作系统源码，自行决定操作系统任务的创建和各任务初始化状态的设置。

- 两种任务调度方式

TOS 分别支持任务级的上下文切换和中断级的上下文切换两种任务调度方式。这两种调度方式在 TOS 的执行过程中非常普遍，前者发生在系统服务中，后者发生在时钟中断的服务程序中。

- 内存分区管理机制

TOS 把连续的大块内存进行分区管理，每个分区中包含整数个大小相同的内存块，但不同分区之间的内存块大小可以不同。

- 可裁剪

TOS 的实现采用高度模块化的方式，用户可以自由定制 TOS 的具体功能，使用部分系统服务，比如用户针对多任务管理、虚拟内存管理、网络协议栈、IO 系统接口、设备驱动、文件系统、以及 BSP 类型等方面分别进行配置，然后根据配置信息生成包含用户要求功能的最小的操作系统镜像。

2.1.2 TOS 的内核结构

TOS 内核主要的工作是对用户的任务进行调度和管理，并且负责管理任务之间的共享资源。TOS 内核所包含的模块主要有任务管理、时间管理、内存管理、任务间通信与同步和任务调度等。TOS 内核体系结构如图 2-1 所示：

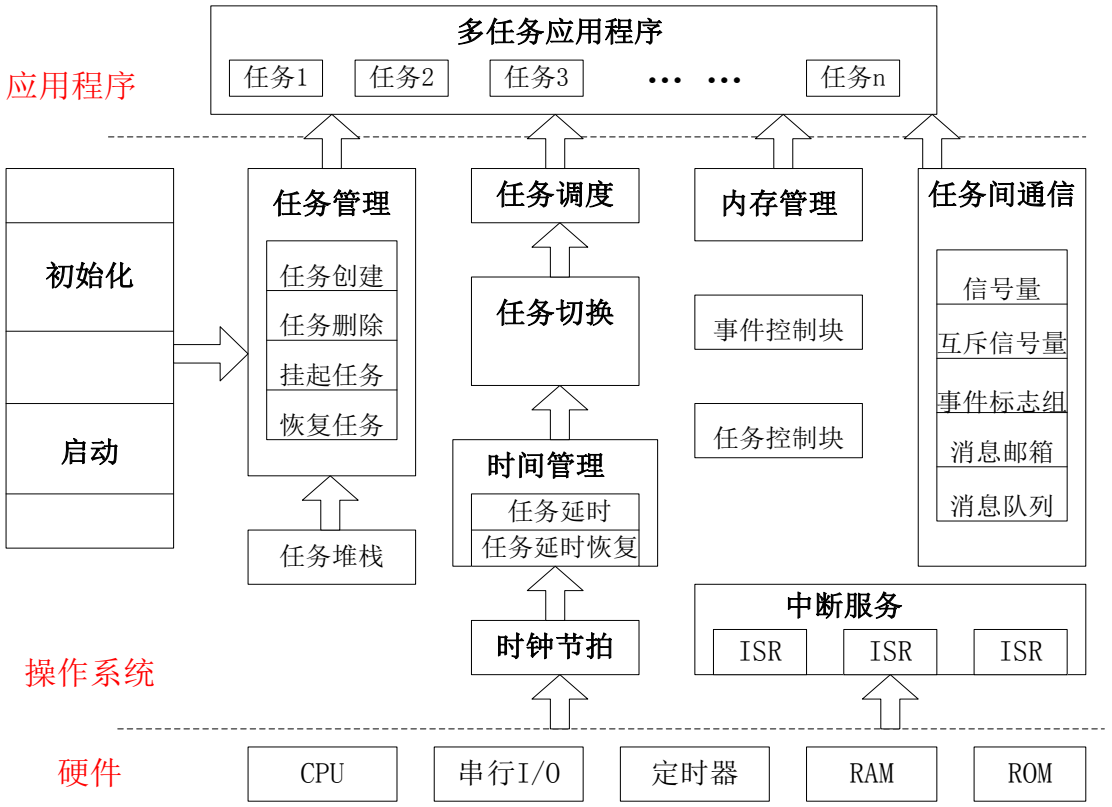

图2-1 TOS体系结构图

2.1.3 主要功能模块

2.1.3.1 任务管理

在现实中人们遇到一个比较大而复杂的问题时，往往使用“分而治之”的方法，将这个大问题分解成为很多个相对简单的小问题，逐个解决这些小问题，大问题也就随之解决了。同样，在面对一个复杂的应用程序时，我们也通常把这个大程序拆分成多个相对较小的任务，然后在计算机中执行这些小任务，最终达到

完成这个复杂的应用程序的目的。由于这种方法可以使计算机系统并发的处理多个任务，从而可以提高计算机系统的利用率，加快了程序的执行速度，因此现代的计算机系统多为多任务操作系统。TOS 就是一个能对这些小任务进行运行管理的多任务操作系统。

（1）任务分类

TOS 的任务有两种：系统任务和用户任务。有系统提供的任务叫做系统任务；由程序设计者编写的任务，叫做用户任务。系统任务是为应用程序服务或为系统本身服务的；用户任务是为了解决应用程序问题而编写的。目前在 TOS 中支持任意多个任务的创建，TOS 中的两个任务类的详细说明如下：

- 系统任务

TOS 预先定义了两个系统任务：空闲任务和统计任务。其中每个应用必须使用空闲任务，而应用程序则可根据自身实际需要来选择使用统计任务。空闲任务和统计任务是在系统初始化时自动产生的。

1）空闲任务

操作系统中的任务有 5 种状态，可是系统很有可能在某个时间点没有用户任务可运行而处于空闲状态。为了使处理器在没有用户任务执行的时候有事情可做，TOS 创建了一个叫做空闲任务 Idle 的系统任务。这个空闲任务除了对系统定义的一个整型变量做累加运算之外，其他几乎什么都不做。当然，若用户认为有需要，也可以在空闲任务中编写一些工作代码。

空闲任务的优先级最低。任何用户应用程序必须使用这个空闲任务，而且通过程序不能删除这个空闲任务。

2）统计任务

统计任务在操作系统中的优先级为次低，此任务负责统计目前系统 CPU 的使用率，方便应用程序了解 CPU 使用率。

不同于空闲任务的必须使用，用户可以根据程序的需求来进行选择是否使用统计任务，当用户需要对 CPU 的调度情况和使用情况进行分析的时候，可以开启统计任务，否则，在普通的任务运行和调度过程中，可以不使用统计任务，以提升真正需运行任务的执行效率。

- 用户任务

TOS 中应用程序的用户任务的创建流程如图 2-2 所示：

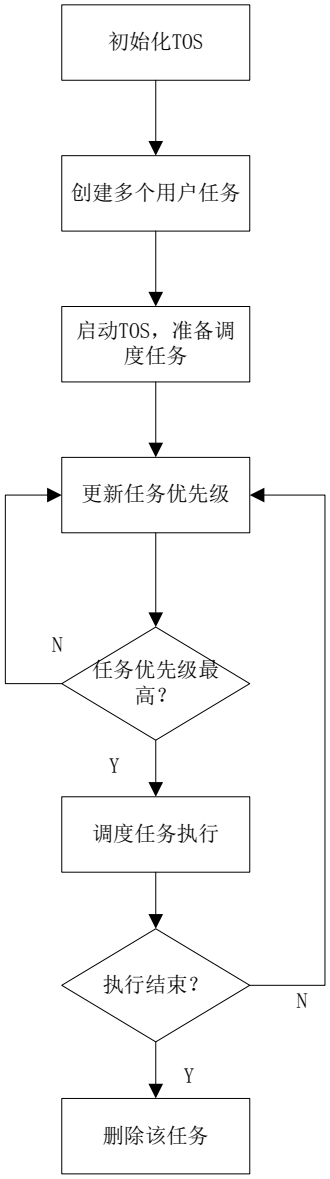

图2-2 TOS的用户任务创建流程图

TOS 中可以支持任意多个任务，这些任务可由用户自行进行初始化，配置包括任务名、任务优先级等任务块初始信息。

（2）任务状态

在 TOS 中，任意一个时刻只会有一个任务拥有 CPU 的使用权，处于运行状态，其他的任务就只能处于其他状态。根据具体情况，TOS 的任务状态被划分为下表 2-1 所列的 5 类。

表2-1 TOS任务状态说明表

任务状态	说明
Ready	系统已为该任务配备了任务控制块，并且已经为此任务在任务就绪表中进行了就绪登记，则此任务就具备了进行运行的充分条，这是任务处于的状态就叫做 Ready 态。
Run	处于 Ready 态的任务如果被调度器给予了 CPU 的使用权，则任务就进入了 Run 态。在一个 CPU 的情况下，任何时候只有一个任务处于 Run 态。
Suspend	一个正在运行的任务若响应中断申请，则会中断运行转而去执行中断服务成型，这是的任务状态叫做 Suspend 态。
Delay	正在运行的任务，需要等待一段时间或等待一个事件的发生再运行时，此任务就会把 CPU 的使用权让给其他任务，接着进入 Delay 态。
Pend	任务没有被分配给任务控制块或者被剥夺了任务控制块是的状态就是任务的 Pend 态。

2.1.3.2 时间管理

TOS 用硬件定时器产生一个周期为毫秒级的周期性中断来实现系统时钟。此中断通过调用中断服务程序 OSTimeTickISR()来完成功能。

由于嵌入式系统的任务是无限循环执行的，并且 TOS 是一个根据优先级的抢占式内核，因此为了防止高优先级的任务长期独占 CPU，给其他优先级别较低的任务获得使用 CPU 的机会，TOS 规定：空闲任务之外的所有任务必须在任务中的特定位置调用 OSTaskDelay()函数，使当前任务延时运行一段时间进行一次任务调度，让出 CPU 的使用权。

进入延迟的任务被挂到 sleep 队列中，等待延迟结束，并启动任务调度。

待到此任务延迟时间结束，则将此任务从 sleep 队列中取出，判定当前任务状态，进入相应任务处理队列中即可。

2.1.3.3 内存管理

在嵌入式系统中，如果多次使用 ANSI C 中提供的 malloc 和 free 两函数来动态分配和释放内存的话，则会导致内存碎片。为了防止这种现象的发生，TOS 采用了内存分区的方法进行内存管理。

在 TOS 中对连续的大块内存进行分区，每个分区中包含整数个大小相同的内存块，但不同分区之间的内存块大小不同。当用户需要动态分配内存时，操作系统就选择一个适当的内存分区，按分区内的内存块分配内存。释放内存时将此块

内存放回它所属的分区，这样就有效地解决了碎片的问题。

2.1.3.4 任务间通信与同步

TOS 中计划使用 4 种同步方式：信号量，消息邮箱，消息队列和事件。

信号量由两部分组成：信号量计数器和等待任务表。信号量计数器是一个 16 位无符号整形值 OSEventCnt；等待任务表是由等待该信号量的任务组成的数组 OSEventTbl[]。

当有任务申请信号量时，若计数器 OSEventCnt 的值大于 0，则计数器 OSEventCnt 的值减 1 并使任务继续运行；若计数器 OSEvnetCnt 的值为 0，则将任务放入等待任务表 OSEvnetTbl[]中。如果有任务释放了正在使用的信号量，则从此信号量的等待任务表 OSEventTbl[]中选择优先级最高的任务放入就绪队列，并启动一次任务调度；如果此信号量的等待任务表 OSEventTbl[]为空，没有等待此信号量的任务，则对计数器 OSEventCnt 加 1。TOS 中关于信号量的处理函数说明如表 2-2 所示。

表2-2 信号量处理函数说明表

函数	说明
OSSemCreate()	建立信号量，并赋予此信号量初值。
OSSemDel()	删除信号量
OSSemPend()	等待信号量
OSSemPost()	发出信号量
OSSemAccept()	无等待的请求一个信号量
OSSemQuery()	查询某个信号量的当前状态

在目前的 TOS 操作系统中，主要以信号量的同步机制为主，对于其余三种同步机制，在本文中暂时不做详细介绍。

2.2 操作系统的任务调度机制

2.2.1 TOS 任务调度流程

在 AT697 上运行的操作系统 TOS 为实时多任务操作系统，任务级图形化分析工具将基于实时操作系统的任务调度函数完成相应的记录、展示和分析。

TOS 对外提供了一套完整的任务调度函数，图 2-3 为一个任务生命周期中的调度过程示意图。

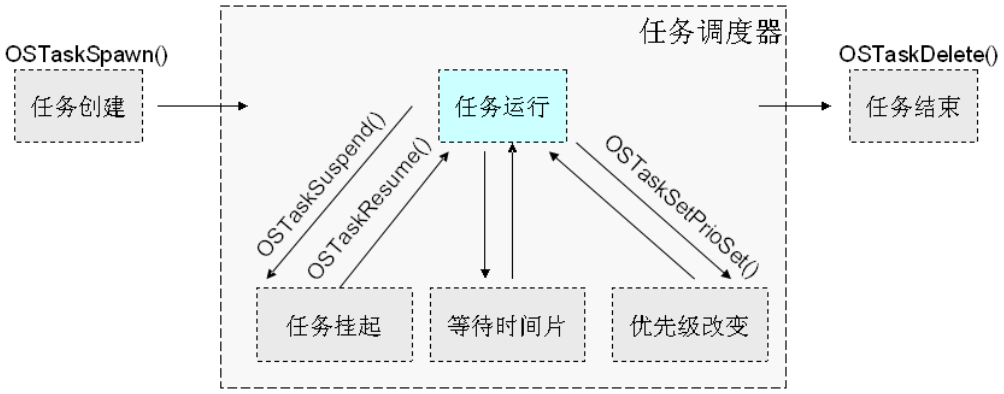

图2-3 TOS任务调度过程

任务从 OSTaskSpawn()开始，如果内存满足堆栈、任务块的新建任务需求时，任务正式创建。在整个任务的生命周期中，其中会经历任务挂起 OSTaskSuspend( )，任务恢复 OSTaskResume( ), 任务延时 OSTaskDelay( ), 任务优先级改变 OSTaskPrioSet()等过程，最终任务删除 OSTaskDelete()，完成整个任务流程。

由于 TOS 是一个可剥夺型实时多任务内核，此内核在任何时候都会运行就绪队列中优先级别最高的任务。TOS 的具体调度状态转换如图 2-4 所示：

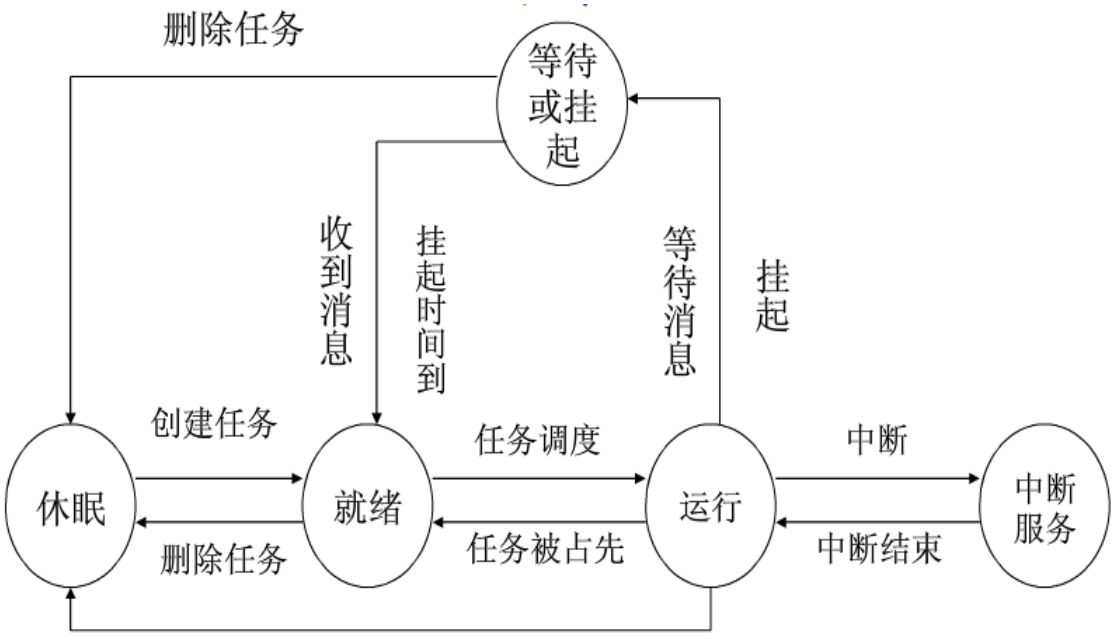

图2-4 TOS任务调度图

TOS 的任务调度是完全基于任务优先级的抢占式调度，若具有最高优先级的

任务处于就绪状态，则此任务就会立即抢占正在运行的任务的 CPU 资源。TOS 中所有任务的优先级均不同，因此任务的优先级也可唯一标识此任务。

任务调度发生的情况有如下两种：

（1）正在占用 CPU 的高优先级任务运行过程中需要某种临界资源，主动提出挂起请求，让出 CPU，此时进行任务调度，从就绪队列中选择最高优先级的任务，使其占用 CPU 进行执行，这种调度称为任务级的上下文切换。

（2）正在占用 CPU 的高优先级任务运行过程中时钟节拍到来，在时钟中断服务处理程序中，内核发现有高优先级的任务已满足了执行条件，则直接切换到高优先级任务执行。这种调度称为中断级的上下文切换。

这两种调度方式在 TOS 的运行过程中十分普遍，前者一般发生在系统服务中，后者一般发生在时钟中断服务处理程序中。

而对于任务信息记录来讲，该操作发生在任务的整个生命周期中，当任务成功创建时，任务控制块结构体记录开始，通过底层函数，其内容会被完整的记录在硬件的系统内存中；当出现有任务的状态变化(包括时间片轮转变化)，其任务信息也将会被记录在系统内存中。当操作系统运行程序空闲或者结束时，将通过串口将之前记录的数据输出的宿主机中，宿主机根据得到的数据完成分析和界面生成的功能。

2.2.2 TOS 的底层函数支持

TOS 为 AT697 板载的多任务实时操作系统，为了完成任务的创建、删除等任务基本操作或多任务的轮时间片切换，条件唤起等任务调度策略，TOS 为任务的调度提供了全方位的函数支持。下表 2-3 为 TOS 中任务管理的基本函数。

表2-3 任务管理基本函数说明表

功能	函数原型
发起一个任务	TASK_ID OSTaskSpawn(U32 name, U32 prio, U32 stacksize, FUNCPTR entry, U32 option, I32 arg1,I32 arg2,I32 arg3,I32 arg4);
删除一个任务	STATUS OSTaskDelete(TASK_ID tid);
任务挂起	STATUS OSTaskSuspend(Task_ID tid);
任务恢复	STATUS OSTaskResume(Task_ID tid);
延时当前任务	STATUS OSTaskDelay(U32 ticks);
重启任务	STATUS OSTaskRestart(TASK_ID tid);

调度上锁	Void OSTaskLock(void);
调度解锁	Void OSTaskUnLock(void);
任务级的任务调度	Void OSTaskSched(void);
任务级的重调度	Void OSTaskReSched(void);
改变任务的优先级	Void STATUS OSTaskPrioSet(TASK_ID tid, I32 newprio);

此外，在 TOS 中还实现了最基本的任务记录功能。见下表 2-4 所示。

表2-4 TOS的基本任务记录函数

功能	函数原型
调度信息记录使能	Void OSTaskInfoRecStart(void);
调度信息记录禁止	Void OSTaskInfoRecStop(void);
调度信息记录	Void OSTaskInfoRec(void);
清空调度信息缓冲区	Void OSTaskInfoRecClear(void);

上述函数可以用来控制操作系统中对于任务状态信息的记录，当需要进行任务分析时，需要通过上述接口将操作系统中对应功能打开。

2.2.3 任务信息记录项

在 TOS 操作系统中，我们使用一个结构体 TCB（Task Control Block）来存储任务相关的所有状态信息，这个结构体包含了任务块的基本信息，主要包括如下所示的 24 项：

- 任务有效标识，
- 任务名，
- 所有任务队列节点，
- 任务当前优先级，
- 任务初始化优先级，
- 任务状态，
- 任务选项，
- 任务函数入口地址，
- 任务参数，
- 任务堆栈大小，
- 任务堆栈基址，
- 轮转剩余时间片，
- 延时时间，

- 睡眠队列中差分延时时间，
- 任务整点上下文，
- 任务浮点上下文，
- 睡眠的队列节点，
- 信号量或消息阻塞队列节点，
- 同优先级任务队列节点，
- 占有的互斥信号量，
- 等待的信号量，
- 等待的消息队列，
- 存放消息的内存缓冲区指针，
- 存放消息的最大字节数

操作系统在运行过程当中，会将上述任务相关信息自动写入到特定的系统内存地址空间当中。为了为用户提供详实的 TOS 任务运行状态变化情况，在实现任务分析机制时，需要完整监测记录的任务块。

在上述 24 项基本信息之中，不同的任务项包含有不同的信息项，某一任务项所包含的信息项可能是一个实数值，也可能是一组实数值。比如任务名只记录一个实数值用来标识任务的名称，而所有任务队列节点这一项就包含了 4 项信息项。通过对任务块 24 项基础信息的分析，我们可以细化得到 90 项的基本记录项，这 90 项基本记录项的具体数据类型归类如下表 2-5 所示：

表2-5 TOS中任务项数据归类表

编号	任务项	说明	大小
1	TASK_ID	任务有效标识	1 个实数型
2	TASK_NAME	任务名	1 个实数型
3-6	TASK_NODE	所有任务队列节点	4 个实数型
7	TASK_PRIO	任务当前优先级	1 个实数型
8	TASK_INIT_PRIO	任务初始化优先级	1 个实数型
9	TASK_STATE	任务状态	1 个实数型
10	TASK_SELECT_ITEM	任务选项	1 个实数型
11	TASK_IN_ADD	任务函数入口地址	1 个实数型

12-15	TASK_ARGS	任务参数	5 个实数型
16	TASK_STACK_SIZE	任务堆栈大小	1 个实数型
17	TASK_STACK_ADD	任务堆栈基址	1 个实数型
18	TASK_REMAIN_TIME	轮转剩余时间片	1 个实数型
19	TASK_DELAY_TIME	延时时间	1 个实数型
20	TASK_DIF_DELAY_TIME	睡眠队列中差分延时时间	1 个实数型
21	TASK_实数_CONTEXT	任务整点上下文	21 个实数型
22-42	TASK_FLOAT_CONTEXT	任务浮点上下文	32 个 FLOAT 型
43-74	TASK_SLEEP_NODE	睡眠的队列节点	4 个实数型
75-78	TASK_BLOCK_NODE	信号量或消息阻塞队列节点	4 个实数型
79-82	TASK_SAME_PRIO	同优先级任务队列节点	4 个实数型
83-86	TASK_MUTEX	占有的互斥信号量	1 个实数型
87	TASK_WAIT_SEM	等待的信号量	1 个实数型
88	TASK_WAIT_QUEUE	等待的消息队列	1 个实数型
89	TASK_PO 实数 ER	存放消息的内存缓冲区指针	1 个实数型
90	TASK_MAX_SIZE	存放消息的最大字节数	1 个实数型

在 TOS 运行中，上述 90 项基本信息项又各自具有不同的变化特点，总的来讲，其变化特点有很大的差异性，为了比较准确地掌握具体的差异性表现，我们对以上 90 项基本信息项进行运行状态信息统计，并对此进行分析。经过对 TOS 长时间运行中各信息项的监测，我们得到 TOS 的 90 项基本信息项的变化情况大致可以分为以下几个不同的特点：

- 大量信息项运行中为恒值

在所有记录的 90 项基本信息项中有 70 项信息在开始运行时赋予初值之后，其在运行中值是不再改变的。在这些为恒值的项中有 52 项信息恒为 0，而其余 18 项恒为初始指定的非零实数值，不管是为 0 还是其他恒值，在进行任务信息记录时都可以进行适当优化以减少记录空间。

- 部分信息项运行中改变

在所有记录的 90 项基本信息项中有 20 项信息在 TOS 的运行中会不断变化，这些变化项需要比较忠实的进行记录其变化值和变化点。

- 部分信息项是在一定范围内取值

对于所记录的 90 项基本信息项中，任务有效标识、任务状态以及任务名等信息项的取值范围是一定的，比如任务有效标识代表了任务当前是否有效，因此只需 0、1 两值即可代表此项信息。

针对上述不同的记录信息变化情况，在进行实时记录时也需要分别采取不同的方法进行处理，以节省空间，提高效率，具体的记录方法我们将在后续的章节中进行详述。

2.3 任务信息的实时记录

从 TOS 的任务调度和实现机制来看，用于保存任务状态信息的那块系统内存空间是我们实现任务分析机制的关键，任务信息的实时显示、实时记录、以及准确重放均依赖于对这部分内存数据的利用。从原理上来讲，为了支持任务的实时显示以及重放机制，有两个问题必须得到解决。

首先，我们必须能够识别出哪些信息是对应任务的状态信息，需要用于后续的任务分析的。具体说来，我们必须能够识别出操作系统什么时候在记录任务信息，以及记录的具体位置在哪里，并且能够截获到这些操作，进行相应的处理。

其次，即使识别到操作系统对任务信息的记录操作，但是这些信息也是实时更新的，新产生的任务信息会马上覆盖掉旧的任务信息，因此，我们需要实时地将这些新生成的任务信息完整地记录到其它地方，以便后续进行重放。

2.3.1 任务信息的识别

通过对操作系统的任务调度机制的分析可以发现，操作系统在运行过程中，会实时地将任务相关信息记录到特定的系统内存中，因此，要对任务信息进行记录的话，必须能够截取到这些写内存的操作，并对写入的信息进行识别和记录。

具体对应到本文的系统中，操作系统运行在软件模拟平台上，那么对这些特定内存的写操作可以由软件模拟器来完成，即由 sim697 模拟器进行操作系统任务信息记录操作的识别和截取。

在这种机制下，sim697 模拟器要做的事情是，识别哪些写操作是对应任务信息的记录，并且需要明确区分出某个写操作对应的是任务信息中的哪一项。由于在 sim697 看来，操作系统对于任务信息的写操作和其他的普通写操作并无区别，都是通过普通的访存指令完成，因此，sim697 要能够准确识别出任务信息的记录操作的话，就需要 TOS 开放任务信息记录的地址以及每个地址和具体任务信息项的对应关系，然后，sim697 模拟器通过监控这一特定地址空间的写操作来检测任

务项信息记录操作，并且将这些任务项信息记录到文件中。

具体实现时，Sim697 在模拟访存写操作时，针对每个写操作的地址进行识别，如果发现属于任务信息记录的特定地址空间，那么进行进一步的分析处理，识别出其对应的是哪一个具体点的任务信息，并进行记录。

除了针对任务信息进行记录外，本文所提机制还会针对全局变量以及中断信息等进行记录。对于全局变量的记录来讲，其原理和对任务信息的记录原理类似，TOS 需要将全局变量的地址信息开放给 sim697 模拟器，以便模拟器能够捕捉到针对全局变量的修改操作。

而对于中断信息，则不需要操作系统的额外支持，所有的中断操作都会通过中断控制器，而这个结构是实现在 sim697 模拟器中的，模拟器通过查看这个中断控制器就能知道所有的中断信息，包括中断号、中断发生的时间等。

2.3.2 任务信息的记录

操作系统运行过程中，会不断的更新任务信息，并且任务信息都是写在固定的系统内存中，当有新的任务信息产生时，会直接覆盖掉旧的任务信息。为了支持后续的任务重放，系统需要将这些实时产生的任务状态信息进行记录。

在记录这些信息时，有两种选择，一是直接记录原始的访存信息，二是先转换成易于识别的任务项信息再记录到文件。由于重放时是指定重放条件的重放，因此不是所有任务项信息都要重放，目前我们选择第一种方法，直接记录原始访存信息，以减少记录时的时间开销。在这种情况下，记录的格式为以下四项信息：

- 访存发生的时间
- 访存的地址
- 访存的数据
- 访存的数据大小

上述 4 项信息包含了任务状态记录的完整信息，后续在重放时，可以基于这些记录再转化成相应的任务状态信息进行重放。

在操作系统运行过程中，任务信息会不断地产生，如果长时间运行的话，将导致记录所需的空间非常大，并且，记录文件过大既会影响到实时记录的速度，同时还会影响到后续重放的速度。因此，系统需要针对记录文件大小进行优化。在本文中，我们提出一些有效的压缩算法，针对任务记录信息进行优化记录。

2.3.3 任务信息记录精度

因为 AT697 内存空间的限制，操作系统用于记录任务信息的空间是有限的，

因此，我们需要根据特定的需求调节记录的精度，精度越高的话，单个任务数据所需的内存空间越大，因此，所能够记录的深度就会越小。

比如，我们可以简单地划分为全精度和半精度两个记录档次。全精度会详细的记录每个任务的任务控制块(24 项)和切换信息(切换时间，堆栈情况等)。半精度会记录每个任务的任务控制块和任务切换直接相关的数据(任务有效标识符，任务名，当前优先级，任务状态，堆栈基址，堆栈大小)和切换信息(切换时间，堆栈情况)。

因为系统分配给任务级信息记录的空间有限，全精度的数据记录深度会小于半精度的数据记录深度。比方说，分配给任务级分析工具的空间为 640KB,一个全精度数据单元为 256B,而半精度的数据单元为为 64B，全精度的记录数据深度为 3760，而半精度的记录数据深度为 10K 个。

当程序空闲、主任务台发来任务监控请求或者程序结束时，AT697 将主动的将数据记录区的数据输出给宿主机，宿主机可以根据自适应精度的存储格式，也可以生成相应的图形界面进行展示。

2.4 任务信息的实时显示

任务信息的实时显示也是任务分析机制的重要组成部分，通过图形化的任务信息实时显示，可以直观地展示任务的运行状态，有助于用户快速发现任务运行过程中的问题。

2.4.1 前台显示的通信机制

前面在介绍任务信息实时记录中讲到，在 TOS 运行期间，需要把任务控制块信息写入指定内存空间，而 sim697 模拟需要主动监测这一特定内存空间内数据的变化，并将写操作信息进行实时的转存。任务信息的实时显示也是类似，需要在检测到任务信息的变化时，在前台显示界面上进行实时的更新。

和任务信息记录的机制类似，为了让前台界面能够知道任务信息的更新，也需要 sim697 模拟器进行协助，sim697 模拟器在检测到任务状态变化时，需要将变化信息提交给前台显示界面，而模拟器和前台界面交互的方式可以有很多种，比如模拟器打印输出，而前台界面通过监控模拟器的打印输出来获取，或者模拟器和前台界面通过一些进程/线程间通信机制[30]来完成。

2.4.2 实时性的保证

可以看到，任务状态信息的实时显示需要消耗一定的计算资源，比如需要将

原始的访存地址信息转化为对应的任务状态信息，sim697 模拟器需要频繁地和前台显示界面进行通信。如果这些操作无法及时的完成，那么就会影响到前台界面的显示速度，当模拟器和前台界面的速度严重不匹配时，就可能导致两者之间的通信消息丢失，继而导致前台显示信息不完整。

为此，我们在 sim697 模拟器和前台显示软件之间引入一个同步机制，这里称之为“窗口同步机制”。具体方式如下：用户预先定义一个同步窗口大小，窗口定义了模拟器和前台显示软件之间可以偏差的最大时间值。当两者运行的时间偏差超过窗口值时，运行更快的需要暂停等待。在我们的系统中，前台显示的任务信息依赖于后台模拟器，因此，前台显示软件不可能运行超过模拟器，这样当两者不匹配时，运行在前的总是后台模拟器，因此，当模拟器当前运行到的时间相比前台任务显示到的时间要超出窗口值时，我们让模拟器需要窗口时间，从而保持两者的时序一致性。

为了避免前台显示速度过慢，我们需要尽可能加速显示功能，为此，我们在前台显示界面中提供一个可配置的功能，当对显示速度要求很高的场景下，我们可以只显示核心的任务状态信息，以保障显示的实时性。

在将任务信息显示出来之前，这里同样还面临一个信息转换的问题，模拟器直接捕捉到的是普通的访存信息，在显示之前，需要将这些信息转化为任务状态信息。这个过程可以由 sim697 模拟器来做，也可以由前台显示界面来做。在我们的系统中，由于显示界面是可以配置的，也就是说，用户可以自由定制在界面下需要显示的任务状态信息，因此不是每个任务状态信息的变化都需要完整的显示在界面上，只有用户选定了的才需要。基于这一考虑，我们将这个转换操作留给前台显示界面来做，后台模拟器只是简单的将原始的访存信息记录发送给前台显示界面，由后者来决定需要接受哪些信息，然后进行相应的转换并显示。

2.5 任务信息的重放

前面我们已经介绍了任务分析机制中两个主要的功能：实时记录和实时显示。有了这两个功能，用户已经可以比较直观的看到任务运行的完整过程，但是对于任务分析机制来讲还不够，当我们发现错误时，更多的情况下是需要往前追溯找到错误的原因，因此，任务信息的重放功能就显得非常重要了。

在重放时，用户可以设置不同的重放条件，系统根据用户设置条件从任务信息记录中进行查询，将符合重放条件的任务信息进行重放显示。目前我们主要提供三种重放条件的设定：

（1）特定时间点

用户可以设定需要回放的时间范围，通过给定起始的指令数，以及终止的指令数进行回放。

（2）特定中断发生点

用户可以设定某个特定的中断类型，然后重放系统重放该中断发生时的临近时间段内的记录信息。

（3）特定变量值变化点

用户可以设定某个特定变量的值为重放条件，即当某个变量的值变化成某个特定值时，我们对变化时的临近时间范围内的记录信息进行重放。

2.6 本章小结

本章主要介绍了嵌入式操作系统任务分析机制的三大主要功能：任务信息的实时记录，任务信息的实时显示，以及任务信息的重放。其中，任务信息的实时记录是整个分析机制的关键，也是后面实时显示以及重放的基础。在下一章中，我们将基于本章所介绍的实现机制，详细介绍任务分析机制的原型软件系统的实现。

第三章 任务分析机制的软件系统实现

为了验证本文所提的任务分析机制，我们基于 sim697 模拟器搭建了一个软件原型验证系统。该系统主要包括两个部分，一是后台的 sim697 模拟器，用于模拟 AT697 处理器硬件平台，二是集成的开发环境界面。我们的任务分析机制和模拟器以及 IDE 界面均有关联，在本章，我们不详细介绍 sim697 模拟器以及 IDE 界面本身的实现，而主要针对任务分析机制相关的三个核心模块进行描述。

3.1 任务分析软件系统介绍

3.1.1 开发环境

本文所实现的任务分析软件系统，是在 Linux 环境下采用 C 语言进行开发完成的，图形化分析工具使用 C++描述的 Qt 跨平台库开发，软件的运行环境为普通的 PC 机，其具体的软硬件要求如表 3-1 所示。

表3-1 系统开发的软硬件环境

规格	说明
操作系统	Linux（开发平台内核版本为 2.6.18）
编译环境	Gcc（开发平台版本为 4.1.2）
调试工具	Gdb
Qt 库	Qt4.7.1
硬件平台	X86_64 指令架构服务器

图形化分析工具软件通过 Qt 提供的 Qt Designer 命令来对界面进行设计（如图 3-2 所示），界面的设计采用模块化的方法，易于扩展，对于添加进来的新功能或新菜单，可以在现有的模块基础上进行快速的实现。

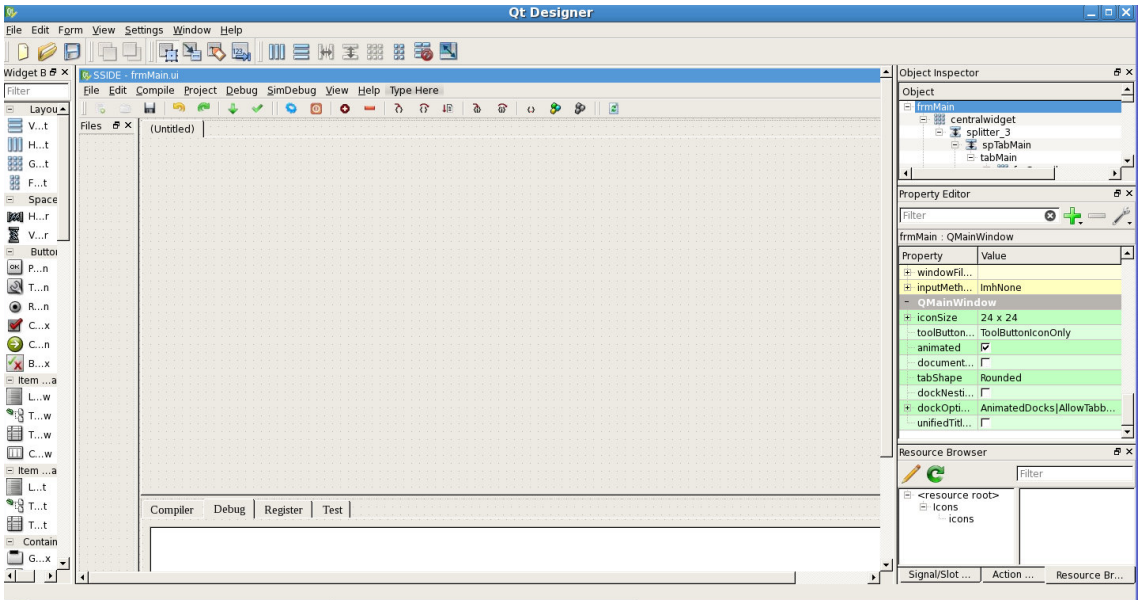

图3-1 Qt Designer软件开发界面图

3.1.2 主要模块构成

基于 sim697 的任务分析软件系统基于软件模拟器和 IDE 界面组成，系统主要包括认务分析界面配置模块、任务分析界面实时显示模块、模拟器性能统计模块、目标系统模拟模块、MI 接口命令处理模块、任务调度信息动态记录模块、任务分析重放模块、以及模拟器运行交互模块等，上述模块之间的关系如图 3-2 所示，这其中主要模块的具体功能如表 3-2 所示：

表3-2 系统主要模块列表

序号	模块名称	模块说明
1	任务分析界面配置模块	本模块在 IDE 界面中为用户提供了操作系统任务级分析所需记录项的选择配置功能，以及操作系统功能模块配置功能。在此模块的支持下，用户可以根据需求和存储空间的限制，手动地选择需要检测的任务项记录。并且根据用户对于操作系统的使用要求，手动配置操作系统运行需要的功能模块。配置完成后，重新编译指定路径下的操作系统，即可生成可以在模拟器上运行的操作系统可执行文件。
2	任务分析界面实时	在 IDE 界面下启动 sim697 模拟器运行并自动加载指定路径下的操作系统可执行文件，而且在 sim697 模拟器运行的过

	显示模块	程中，自动读取输出的任务级记录数据，在界面中显示出来。为用户提供一个直观的图形化界面，完整并直接的展示任务的运行状态，为开发、调试人员提供清晰的任务级查看和分析工具。
3	模拟器性能统计模块	在模拟器运行过程中，针对 sim697 运行过程中的性能数据进行统计。包括处理器内部的实时状态信息，如各寄存器信息、流水线状态信息、各类 Memory 信息等，以及对模拟器上运行的程序的关键性能数据进行统计，如程序特征、缓存访问命中率等运行性能统计信息。
4	目标系统模拟模块	模拟完整的 AT697 处理器结构，除了处理器核之外，还包括常见的片上设备，如计时器、看门狗等。
5	MI 接口命令处理模块	模拟器支持接收第三方工具发送过来的符合 MI 接口规范的操作命令，命令列表可以以文件的方式提供，执行结果支持以打印的方式输出到指定文件。
6	任务调度信息动态记录模块	为了保证任务运行时的信息在任务执行结束之后也能观察到，我们对任务运行过程中的动态信息进行实时动态的记录，以供后续的重放输出。
7	任务分析重放模块	读取任务调度信息动态记录模块生成的动态信息记录文件，并根据用户的重放条件要求，重现操作系统在模拟器上运行时的任务状态信息。
8	模拟器运行交互模块	用于支持 IDE 和 sim697 模拟器之间完成必要的交互功能。

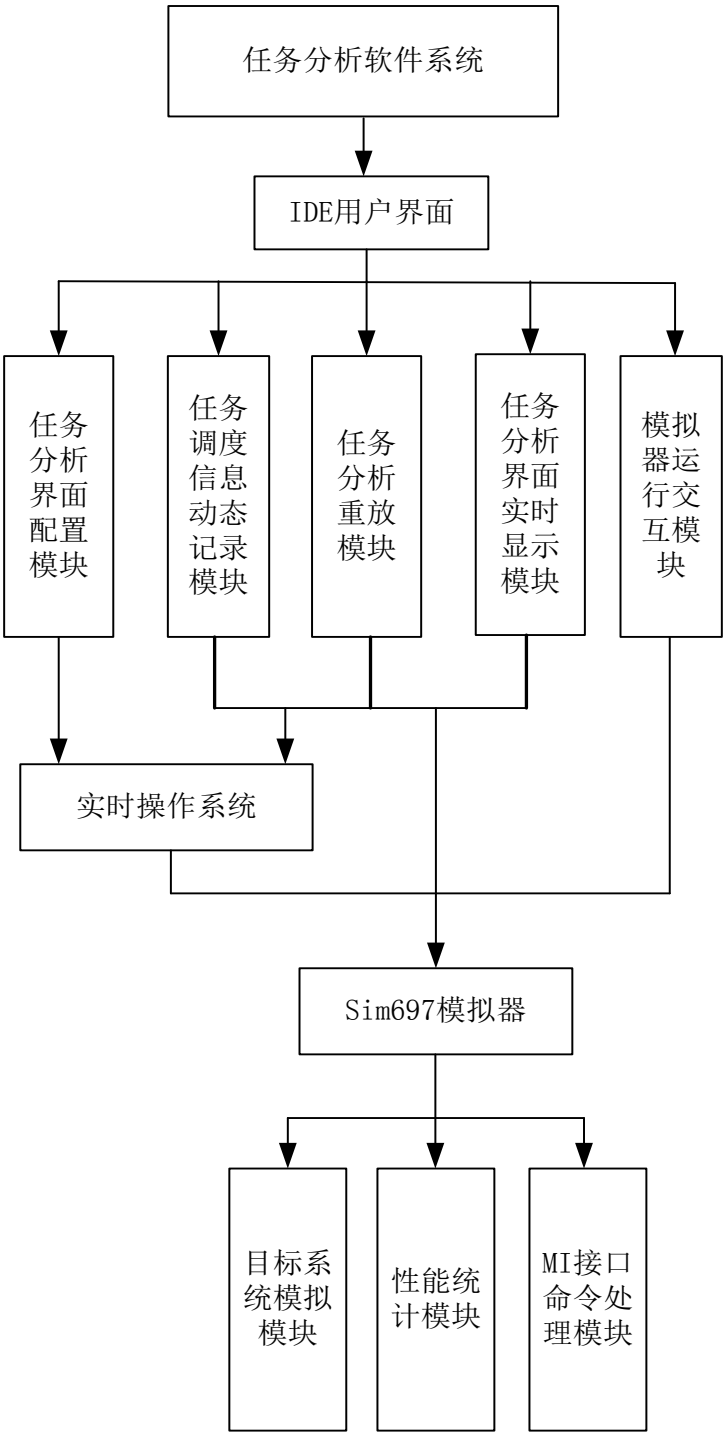

图3-2 系统主要模块关系图

3.1.3 操作界面

基于 sim697 模拟器的实时操作系统任务分析软件界面风格简约舒适，主要包括标题栏、菜单栏、工具栏、工程文件管理区域、源码编辑区域、调试信息显示区域、内存实时动态显示区和字符命令输入区域，如下图 3-3 所示。

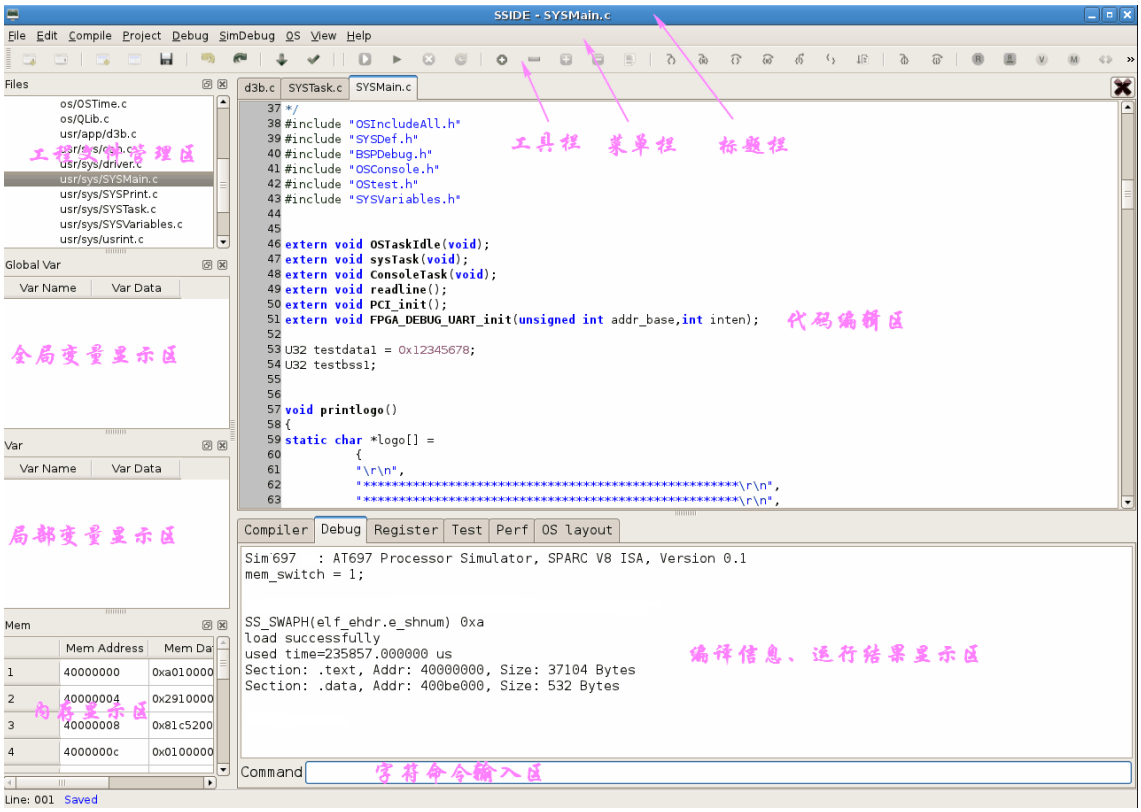

图3-3 软件界面示意图

该操作界面中各部分的主要功能说明如下：

- 标题栏

  标题栏中间显示当前打开的文件名，右边是最小化、最大化和关闭按钮。

- 菜单栏

  菜单栏包括 File、Edit、Compile、Project、Debug、SimDebug、View、Help 等项。File 菜单实现对文件的管理功能，例如新建文件、打开文件、保存文件等；Edit 菜单实现对文件的编辑功能，例如粘贴、复制、撤销、查找等；Compile 菜单实现是对工程的编译、运行等功能；Project 菜单实现对工程文件的管理功能，比如新建工程，打开工程，删除工程等；Debug 菜单实现远程 GDB 调试功能，包括断点设置功能、单步执行功能、继续运行功能、寄存器读写、内存读取功能等；SimDebug 菜单实现集成 sim697 模拟器自身的调试功能；View 菜单包括工具栏的选择，及界面的复位；Help 菜单目的是显示当前界面版本以及帮助信息，目前还未集成。

- 工具栏

工具栏按钮依次包括工程管理相关命令：新建工程、打开工程、保存工程；操作相关命令：撤销、撤销重复；编译相关命令：编译、运行；Debug 调试相关命令：启动、退出、断点插入、断点删除、step、next、continue、stepi、nexti、变量读取、寄存器读取、内存读取、复位按钮。

- 工程文件管理区：

  用于工程与文件的管理，如打开工程、编程工程等。

- 源码编辑区：

  此区域可进行源代码的编辑，除了使用键盘操作之外，还可使用界面提供的“编辑”菜单及其相应的快捷方式进行操作，包括“剪切”、“复制”、“粘贴”等功能。对于显示的源码，支持常用编程语言，支持语法高亮等功能。

- 编译调试信息显示区

  显示当前工程的编译信息，调试信息，运行信息，寄存器信息等，除此之外，程序运行的字符输出信息也将显示在这里。

- 内存实时动态显示区：

  实时动态显示当前模拟器内存值。

- 字符输入命令区

  供用户手动输入调试命令。

借助上述操作界面，在本软件系统中，用户可以实现实时操作系统的编辑、编译、运行、调试等功能，操作系统任务分析相关的功能也将借助于此界面进行交互和显示，在后面的几个小节中，本文将重点介绍和任务分析机制相关的几个部分的具体实现机制。

3.2 实时记录功能

3.2.1 数据结构定义

上一章中介绍了任务信息所包含的内容，在系统实现时，为了方便任务信息的记录，我们定义了对应的数据结构用于保存该任务信息，其内容如下所示：

```
struct task_info{
        TASK_ID,                    任务有效标识
        TASK_NAME,                  任务名
        TASK_NODE,                  所有任务队列节点
```

TASK_PRIO,	任务当前优先级
TASK_INIT_PRIO,	任务初始化优先级
TASK_STATE,	任务状态
TASK_SELECT_ITEM,	任务选项
TASK_IN_ADD,	任务函数入口地址
TASK_ARGS,	任务参数
TASK_STACK_SIZE,	任务堆栈大小
TASK_STACK_ADD，	任务堆栈基址
TASK_REMAIN_TIME,	轮转剩余时间片
TASK_DELAY_TIME,	延时时间
TASK_DIF_DELAY_TIME,	睡眠队列中差分延时时间
TASK_INT_CONTEXT,	任务整点上下文
TASK_FLOAT_CONTEXT,	任务浮点上下文
TASK_SLEEP_NODE,	睡眠的队列节点
TASK_BLOCK_NODE,	信号量或消息阻塞队列节点
TASK_SAME_PRIO,	同优先级任务队列节点
TASK_MUTEX,	占有的互斥信号量
TASK_WAIT_SEM,	等待的信号量
TASK_WAIT_QUEUE,	等待的消息队列
TASK_POINTER,	存放消息的内存缓冲区指针
TASK_MAX_SIZE,	存放消息的最大字节数
TASK_START_TIME,	任务运行起始时间
TASK_FINISH_TIME	任务结束运行时间

}

前面说过，系统实际运行时并不是一定要记录所有的任务信息项的，用户可以自己配置，因此，在实际使用时，上述结构体中只包含用户选择的配置记录项，用户未选择的项不在结构体中，从而可以减少结构体所占内存空间大小，提高任务记录深度。

此外，为了方便用户更全面地监测系统运行，除了针对任务状态信息的记录，我们还相应增加了对系统全局变量和中断的监测。对全局变量的检测与对任务项的检测类似，系统需要先获知全局变量所在的地址信息，从而实现对这些特定地址的监测。而对于中断的监测，则直接在模拟器中可以获得所有的中断信息，如中断号、中断发生的时间、以及中断发生时对应的任务信息等。对于全局变量以

及中断信息的监测，我们分别使用如下结构体来记录。

struct os_var{

OS_GV_ADD	全局变量始址
OS_GV_SIZE	全局变量的大小
OS_GV_DATA	全局变量的值
OS_GV_TIME	全局变量变化的时间

}

struct os_int{

OS_INTER_TASK_ID	中断务 ID
OS_INTER_ID	中断号
OS_INTER_TIME	中断发生的时间

}

由于全局变量，并不一定放在连续的地址空间中，因此需要在解析目标二进制文件时，把所有的全局变量信息预先解析并记录下来，解析全局变量的具体流程代码如图 3-4 所示：

```
void  Global_Var_Handle()
{
    system("readelf  testos/make/a.out -a > Sys_file/a.info ");

    FILE *Var_fp;
    if((Var_fp = fopen("Sys_file/a.info","r+")) == NULL)
    {
        printf("Cannot open file 'Sys_file/a.info'\n");
        exit(0);

    }
    char *one_line;
    int buff_size = 120;
    int i=0;
    one_line = (char*)malloc(buff_size*sizeof(char));
    begin = (struct Node *)malloc(sizeof(struct Node));

    current = begin;
    end = begin;

    while(fgets(one_line,buff_size,Var_fp)!=NULL)
    {
        char *token = strtok(one_line," ");
        char *val1=NULL;
        char *cmp1="OBJECT";
        char *val2=NULL;
        char *cmp2="GLOBAL";
        while(token != NULL)
        {
            if(i == 1)
            {
                current->addr =strtol( token,NULL,16 );
            }
            if(i == 2)
            {
```

```
                current->size =strtol( token,NULL,16 );
            }
            if(i == 3)
            {
                val1 = token;
            }
            if(i == 4)
            {
                val2 = token;
            }
            if(i == 7)
            {
                char *Name1, *Name = token;
                Name1 = strtok(Name,"\n");
                strcpy(current->glb_var,token);
            }
            if((i==7)&&(!strcmp(val1,cmp1))&&(!strcmp(val2,cmp2))&&
                    (strcmp(current->glb_var,"TaskDeleteAddr"))&&
                    (strcmp(current->glb_var,"TaskAddr"))&&
                    (strcmp(current->glb_var,"OSConfig"))&&
                    (strcmp(current->glb_var,"TaskRunStatus"))&&
                    (strcmp(current->glb_var,"TaskCountpointer")))
            {
                current = (char*)malloc(buff_size*sizeof(char));
                end->next = current;
                end = current;
            }
            i++;
            token = strtok(NULL," ");
        }
        i = 0;
    }
    fclose(Var_fp);
    current = begin;
    if((Var_fp = fopen("Sys_file/Var_All.sys","w"))==NULL)

    {
        printf("Cannot open file 'Sys_file/Var_All.sys'\n");
        exit(0);

    }
    while((current->glb_var != NULL)&&(current->addr != 0x0))
    {
        fprintf(Var_fp,"%s ",current->glb_var);
        fprintf(Var_fp,"0x%x ",current->addr);
        fprintf(Var_fp,"0x%x \n",current->size);
        current = current->next;
    }
    fclose(Var_fp);
}
```

图3-4 从目标文件中解析全局变量

此后，在模拟器执行过程中，动态地将当前访存地址和记录的全局变量地址信息进行比对，以判断当前操作是否是改写全局变量的操作，其比对部分的核心代码如图 3-5 所示：

```
157     /* for Global Variety*/
158     while(Mcurrent&&(sim_num_insn>=1)&&(RecEnable))
159     {
160         if((addr >= Mcurrent->addr)&&(addr <  Mcurrent->addr + Mcurrent->size))
161         {
162             if(Mcurrent != Mcurrentlast)
163             {
164                 Global_Var_Record(Mcurrent);
165                 Mcurrentlast = Mcurrent;
166             }
167             break;
168         }
169         Mcurrent = Mcurrent->next;
170     }
```

图3-5 针对全局变量操作进行比对识别的代码

3.2.2 实时记录的执行流程

在我们的原型系统中，实时记录功能主要是和 sim697 相关，需要在 sim697 的执行过程中实时的截取到任务信息更新操作并进行记录，其执行的流程如图 3-6 所示：

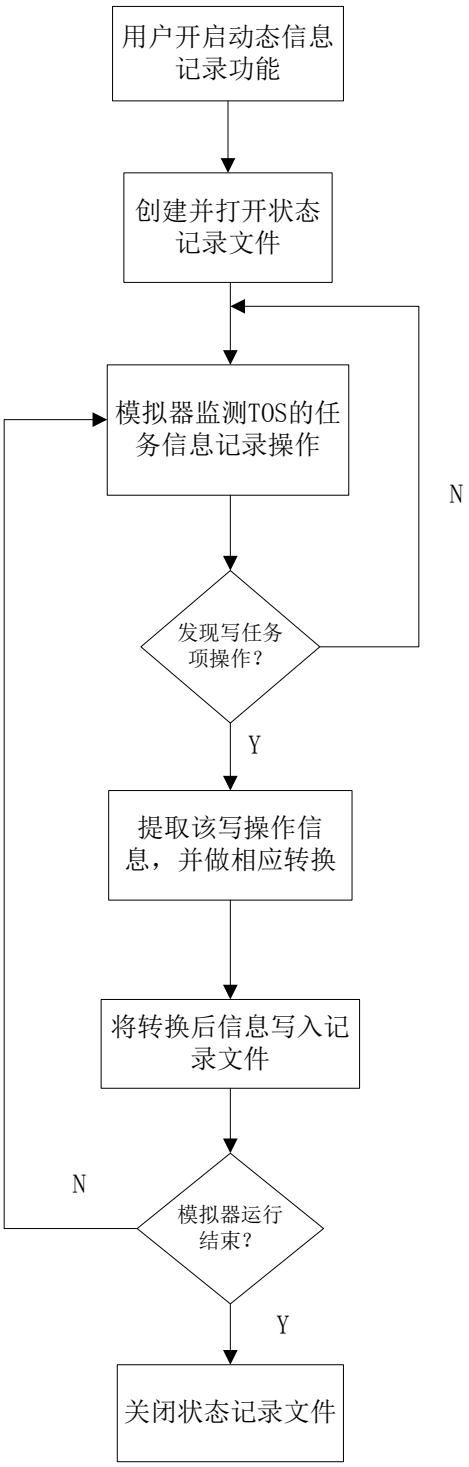

图3-6 实时记录功能执行流程图

当用户启动了状态信息记录功能后，系统会创建并打开一个用于记录任务动态信息的记录文件，然后在模拟器执行过程中监测 TOS 发出的任务信息记录操作，当检测到写操作时，提取出该操作相关信息，并转换成适合存储的格式（该过程可选，也可以直接存储原始信息，在后续使用时再转换），并写入到记录文件。模

拟器在运行结束之前，会一直检测这类任务信息记录操作，并进行动态记录，直到模拟结束，关闭记录文件。

```
239 void Mem_tcb_record( ss_addr_t addr)
240 {
241     int i;
242     for(i=0;i<TCB_COUNT;i++)
243     {
244         if(((addr >= TCB_REC[i])&&(addr <= (TCB_REC[i]+TCB_SIZE))))
245         {
246             int p,q;
247             FILE * tcb_runstat_fp;
248 
249             char *token;
250             mem_access(mem,Read,TCB_REC[i]+0x4,&p,4);
251             int number = (addr-TCB_REC[i])/0x4 + 1;
252 
253             if(number == STATUS_INFO)
254             {
255                 *token = "#1";
256                 if((tcb_runstat_fp = fopen("Sys_file/tcb_runstat_record.sys","a+"))==NULL)
257                 {
258                     printf("Cannot open file 'Sys_file/tcb_runstat_record.sys'\n");
259                     exit(0);
260                 }
261 
262                 fprintf(tcb_runstat_fp,"%d ",sim_num_insn);
263                 fprintf(tcb_runstat_fp,"%s ",token);
264                 fprintf(tcb_runstat_fp,"0x%x ",SS_SWAPW(p));
265                 fprintf(tcb_runstat_fp,"0x%x ",number);
266                 mem_access(mem,Read,addr,&q,4);
267                 fprintf(tcb_runstat_fp,"0x%x ",SS_SWAPW(q));
268                 fprintf(tcb_runstat_fp,"\n");
269                 fclose(tcb_runstat_fp);
270             }
271             else if(number == TRAP_INFO)
272             {
273                 //record trap infomation
274             }
275 
276         }
277     }
278 }
```

图3-7 任务TCB信息记录代码

实时记录的核心是任务 TCB 块信息的记录，模拟器比对访存地址是否位于某个任务的 TCB 块信息所在地址空间，若是，则进行记录，在代码实现中，我们将不同的 TCB 信息记录到不同的文件，比如状态信息记录到 sys_file 文件夹下的 tcb_runstat_record.sys 文件中，如图 3-7 所示，而全局变量信息我们又记录到其他的文件 tcb_record.sys 中，其代码如图 3-8 所示。

```
362 void Global_Var_Record(GNode * GVarNode)
363 {
364     if((tcb_fp = fopen("Sys_file/tcb_record.sys","a+"))==NULL)
365     {
366         printf("Cannot open file 'Sys_file/tcb_record.sys'\n");
367         exit(0);
368     }
369
370     fprintf(tcb_fp,"%d ",sim_num_insn);
371     char *token = "#4";
372     fprintf(tcb_fp,"%s ",token);
373     fprintf(tcb_fp,"%s ",GVarNode->glb_var);
374     fprintf(tcb_fp,"0x%x ",GVarNode->addr);
375     fprintf(tcb_fp,"0x%x \n",GVarNode->size);
376     fclose(tcb_fp);
377 }
```

图3-8 全局变量记录代码

3.3 实时显示功能

3.3.1 实时显示界面

系统运行时，需要先在前台图形界面下启动 Sim697 模拟器运行，并自动加载指定路径下的 TOS 可执行文件，在 Sim697 模拟器运行的过程中，会自动读取输出的任务级记录数据，在界面中显示出来。我们的原型系统为用户提供一个直观的图形化界面，完整并直接的展示任务的运行状态，为开发、调试人员提供清晰的任务级查看和分析工具。

本系统的前台显示界面基于 Qt 图形库[31,32]开发，参考 Eclipse[33]界面实现方式。图 3-9 是操作系统运行时，界面提供给用户的表格型的实时监测界面。监测内容包括任务有效标示符、任务名、任务当前优先级等信息。

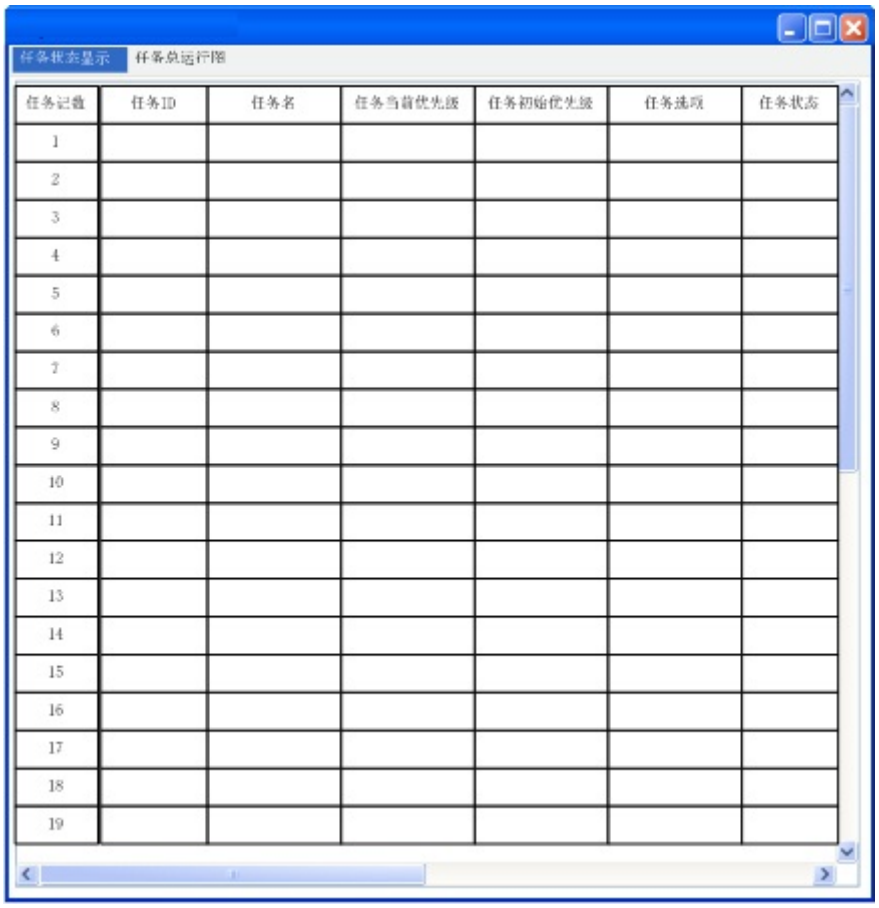

图3-9 OS运行实时监测界面

图 3-10 是操作系统运行过程中总的任务运行状态图，横轴代表运行时间轴，纵轴代表各任务，当任务状态为运行时，便在对应时间点显示出来，若某任务产生中断，则在图中此任务和中断发生时间对应的横纵坐标处，以红点显示出来。

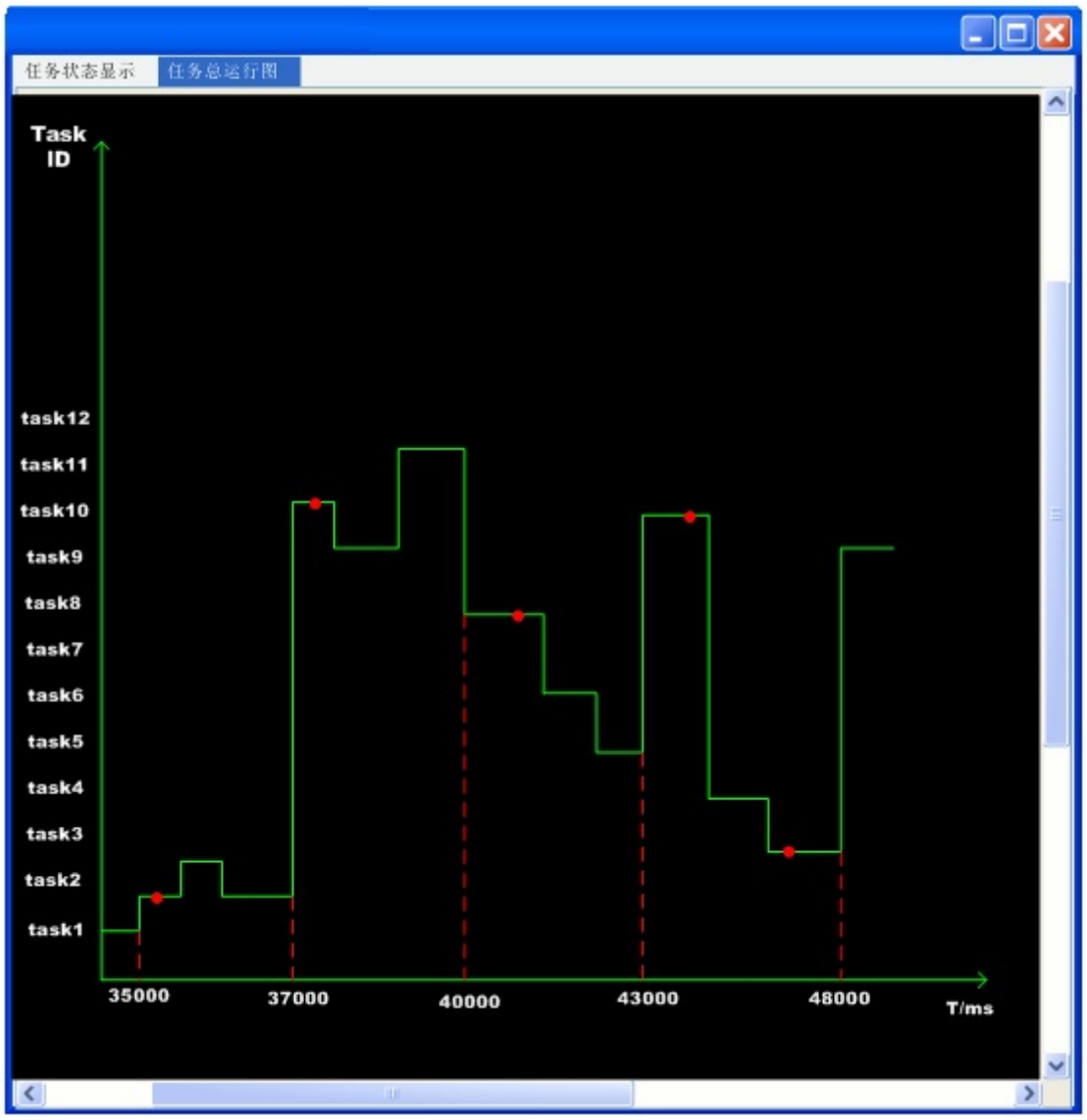

图3-10 任务总的运行状态图

为了方便使用，运行图也实现了显示大小可调节的功能。当用户需要查看当前时间点之前某个任务的状态切换情况，可以左键双击该任务对应的运行图线，此时便会弹出单个任务运行状态转换图，单个任务输出的时间信息可以按照横轴时间，纵轴状态的方式进行图形化的显示，基本的界面情况如图 3-11 所示；单击鼠标右键便会显示该任务的任务名、任务当前状态、其所占有的信号量，以及等待的信号量等信息，如图 3-12 所示。

图3-11 单个任务状态转换图

```
Name: task_name
Current state: Running
Mutex num: xxx
Mutex name:  mutex1,
             mutex2,
             mutex3...
Wait sem:    mutex4
Interrupt num: xxx
...
```

图3-12 任务相关信息示意图

由于 TOS 从修改任务到在前台更新修改内容，信息经过 Sim697 模拟器以及前台显示模块处理显示，第二章说到过，由于模拟器和显示模块是异步运行的，这样前台显示就与后台 TOS 运行存在一定的误差，有可能造成后台传输数据速度大于前台处理显示的速度，直接导致前台存储空间溢出，因此，我们使用“窗口同步机制”来处理此种问题。

应用到我们的原型系统中时，这一操作可以放在模拟器中进行检测，具体的

流程如下：模拟器运行时，自动检测指定地址空间（存储任务项信息的地址空间）是否有写操作，若存在写操作，输出写操作的内容，等待前台处理显示输出，同时检测前台显示进度，当发现模拟进度和模拟器自身运行之间的时间误差大于窗口值时，模拟器将暂停执行，等待前台处理完毕所有的显示项之后，向后台模拟器发送继续执行命令，后台模拟器收到该消息后再接着向下执行。这样，系统通过动态的实时调整，保证了前台显示的实时要求。

需注意的是，如果与此同时，用户启动了重放记录功能，那么系统在运行过程中，模拟器还需要负责将任务状态信息、中断信息、全局变量信息等记录到指定文件，以便于后续的重放操作。

3.3.2 显示信息的配置

系统运行时动态记录任务的全部状态信息会引入很大的时间开销以及空间开销，性能开销方面，针对任务信息生成的检测、记录，实时显示时的绘图刷新操作，后台模拟器和前台显示界面的通信和同步操作等都需要占用运行时间。相比于普通的执行，增加任务信息记录和显示功能后，系统运行速度会变慢。空间开销方面，主要是用于存储的任务状态信息需要占用较大的存储空间，为了支持重放，需要长时间地记录所有任务执行过程中的全部状态变化信息。

为了降低系统运行的开销，本系统提供灵活可配置的记录功能，即系统支持用户针对需要记录和显示的任务状态项信息进行配置，比如，用户可以自行选择记录所有的 24 项任务控制块信息以及切换时间信息，或者只记录任务控制块中的若干关键信息和任务切换信息。用户选择的信息项越少，对执行性能的影响也就越小。

系统提供的配置对话框界面如图 3-13 所示：

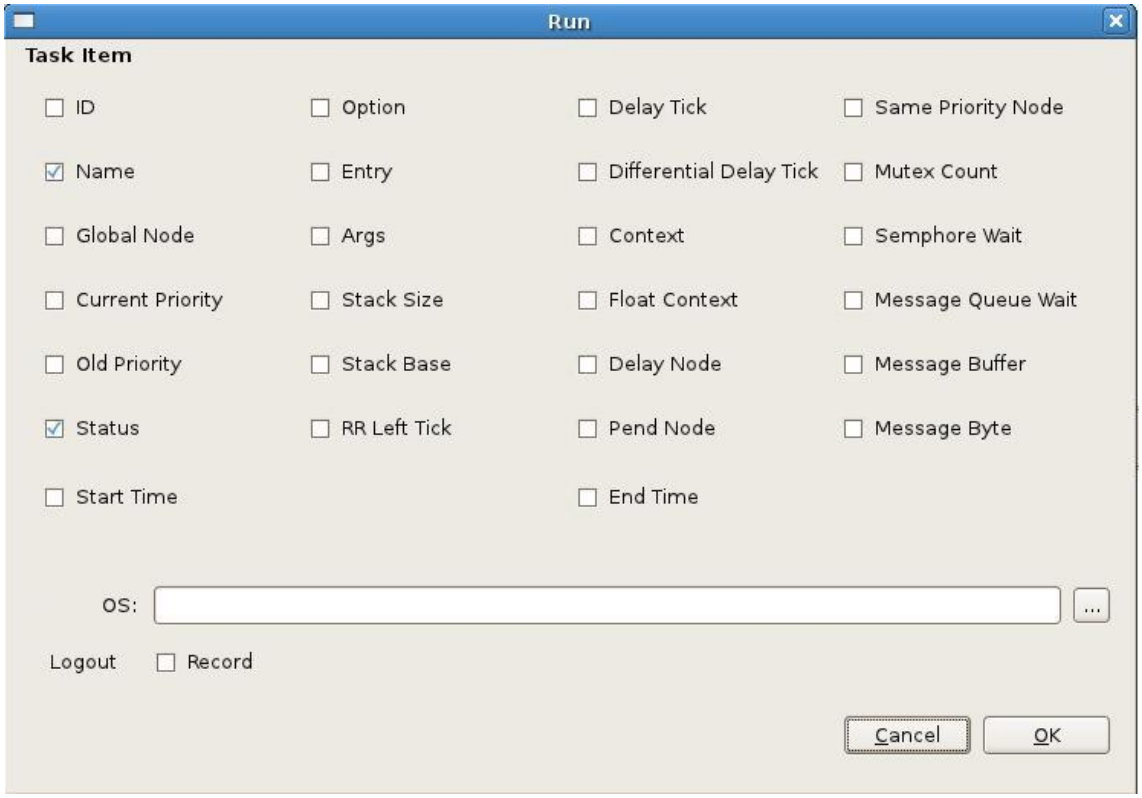

图3-13 任务记录信息配置界面

该对话框提供了所有任务信息项，用户可以根据需求选择需要显示的任务项，然后指定操作系统可执行文件存在的路径，点击“OK”键即可开始执行 OS 的运行和任务信息的动态实时记录和显示。

此外，OS 中实现了任务记录使能和禁止的开关函数，Sim697 模拟器能够支持整体使能或者关闭任务级分析功能，在关闭模式下，本软件不会对上述任何信息进行记录和显示。

系统启动后将自动根据用户配置的记录和显示信息进行记录和显示。图 3-14 为用户指定的任务信息项的实时表格显示，在操作系统运行过程中可以实时更新。

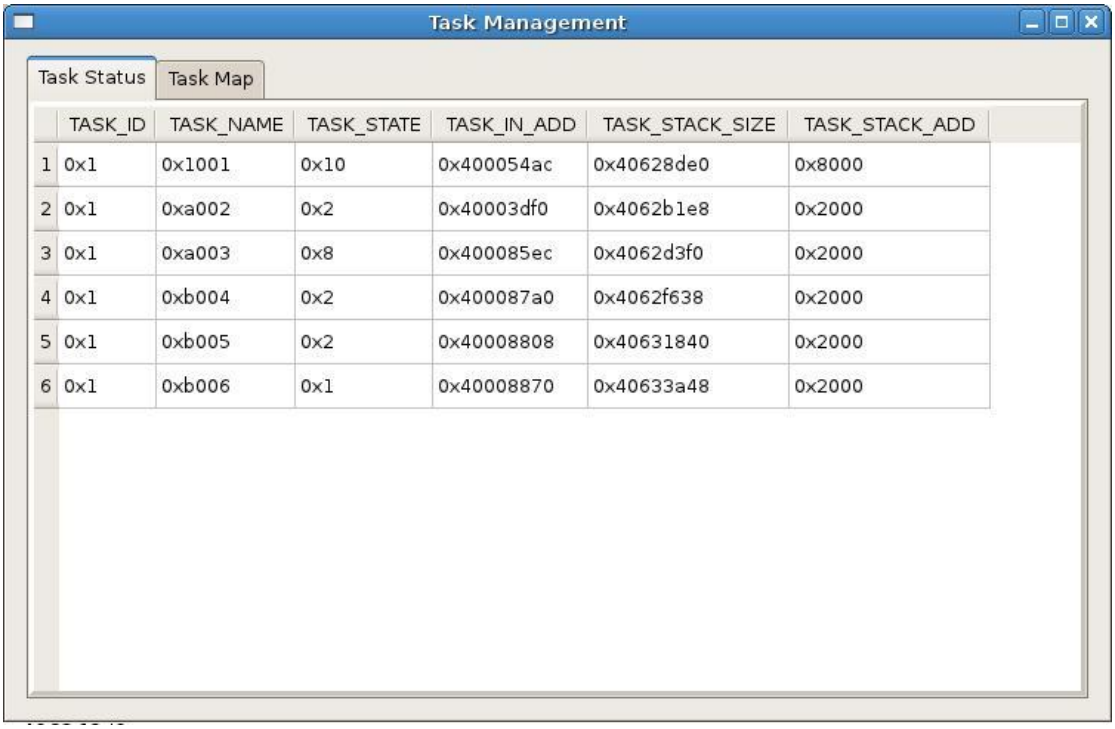

图3-14 任务项表格显示

图 3-15 为任务运行的实时任务运行图，该运行图横坐标是运行的 cycle 数，纵坐标是当前操作系统运行的任务数及对应的任务名。左上角提供的“Start”按钮和“Pause”按钮可以随时暂停或开始运行操作系统，当操作系统处于暂停状态时，可以利用鼠标左键及鼠标滑轮实现运行图的缩放及拖动功能。StartTime 和 EndTime 输入框可以设置需要显示的 Cycle 范围，设置好后，点击“Set Time Range”按钮，以方便查看任务项信息细节。点击“default Time Range”按钮将回到系统默认 Cycle 更新方式。用户可以根据需要选择左下方的“Display interrupt”选择框来显示操作系统运行过程中的中断信息。

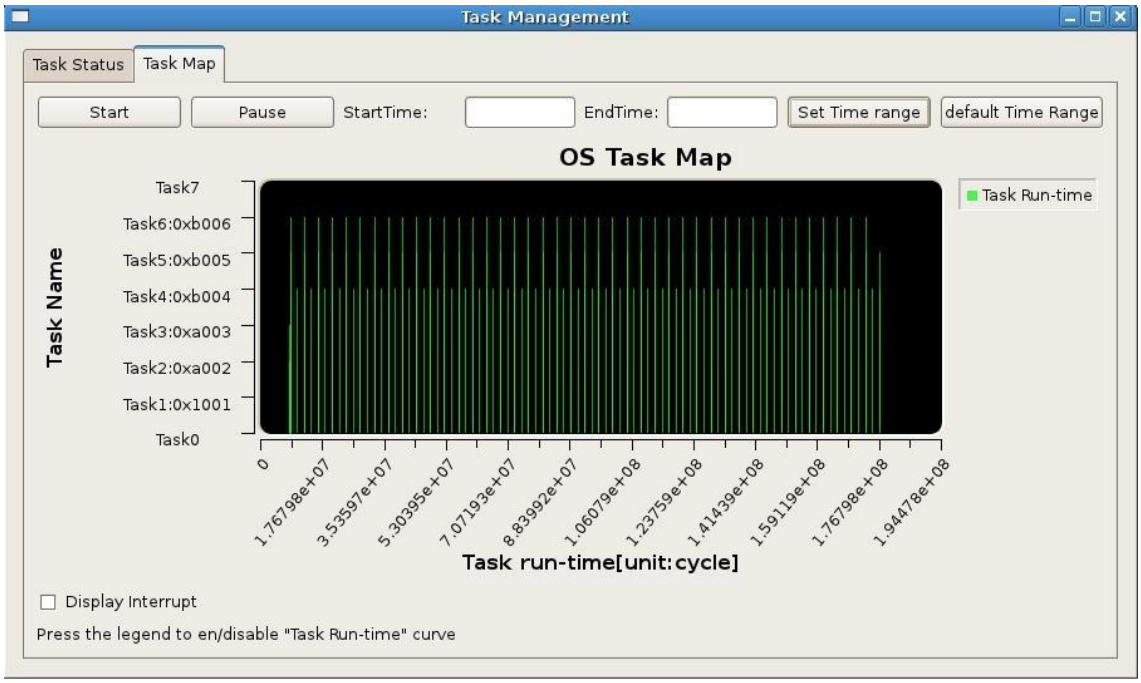

图3-15 任务项图形显示

以纵坐标任务项 ID 为基准，鼠标双击该运行图相应区域可以显示单个任务的任务转换图，如图 3-16 所示。该任务转换图横坐标是相应任务状态运行的 Cycle 区间，纵坐标是任务运行状态，如图 3-12 中任务 1（0x1001）在图示柱状时间段内的状态是 Ready。

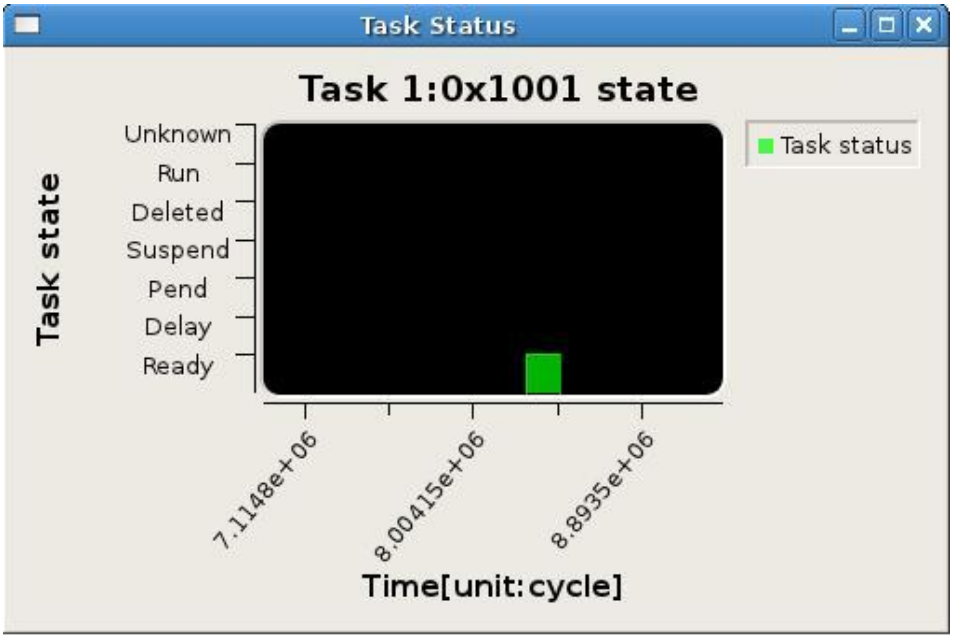

图3-16 单个任务转换图

3.3.3 实时显示的流程

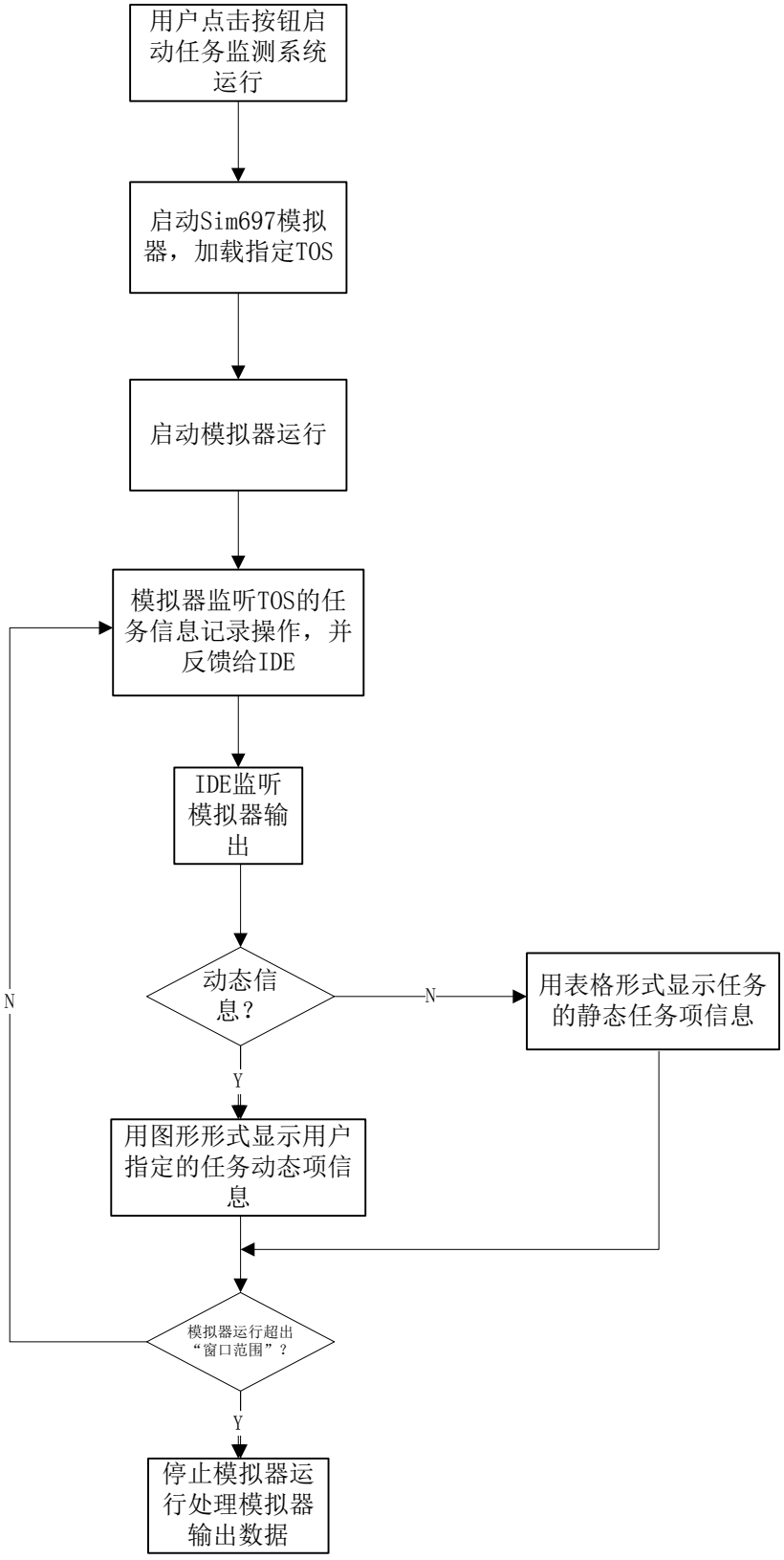

图3-17　实时显示功能操作流程图

实时显示功能的操作流程如图 3-17 所示，任务监测系统启动后，会在后台启动 sim697 模拟器，并自动加载 TOS 运行，TOS 运行过程中，Sim697 模拟器会监听 TOS 的任务信息记录操作，将记录结果通过模拟器输出给 IDE，在 IDE 读取到输出后，根据所显示信息是静态不变的还是动态变化的，在界面中显示出来。需注意的是，这个过程需要保证模拟器和前台显示的同步，当模拟器执行过快，超出限定“窗口”范围时，需要暂停模拟器的运行。

3.4 重放功能

为了方便任务的分析，系统提供任务重放执行功能，为了支持该功能，系统在 OS 运行期间，指定一个文件来记录操作系统运行时模拟器输出的所有实时信息数据。这样当程序执行结束时，用户仍然可以依据这些记录文件信息来重放操作系统运行过程中的所有任务项信息。

3.4.1 重放功能实现

在重放时，用户可以根据之前观测操作系统运行的状况，手动设置重放条件，此模块便根据用户设定的重放条件，寻找满足重放条件的记录信息，并根据这些记录信息恢复出任务的状态变化过程，在界面上予以显示。目前系统提供的是三种重放条件的设定：

（1）特定时间点

用户可以设定需要回放的时间范围，通过给定起始的指令数，以及终止的指令数进行回放。

（2）特定中断发生点

用户可以设定某个特定的中断类型，然后重放系统重放该中断发生时的临近时间段内的记录信息。

（3）特定变量值变化点

用户可以设定某个特定变量的值为重放条件，即当某个变量的值变化成某个特定值时，我们对变化时的临近时间范围内的记录信息进行重放。

为了节省不必要的转换操作，系统在记录这些用于重放的任务信息时，采用直接记录原始访存信息的方式，共包括以下四项信息：

- 访存发生的时间
- 访存的地址
- 访存的数据
- 访存的数据大小

在运行中，为了方便用户更加全面的监测系统运行状况，本系统还增加了两个系统监测项：全局变量监测和系统中断监测。全局变量监测项监测系统中各个全局变量的变化情况，系统中断监测记录系统中产生的各中断的类型以及产生时间等信息。

```
3335      if((sim_num_insn % 1000000 == 0)&&(RecEnable))
3336      {
3337          OS_tcb_update();
3338      }
```

图3-18 定期检测是否需要执行TCB完整信息保存的代码

需要注意的是，由于我们每次只是记录任务信息中的变化项，而不是完整的把整个 TCB 的所有信息都记录下来，因此，在重放时，我们无法直接从用户指定的那个重放时间点开始，因为我们不知道这个点的完整 TCB 状态信息，为了保证重放的正确性，需要从具有完整 TCB 信息的位置开始重放。为了减少重放的额外开销，我们在系统中定期的保存一个完整的 TCB 信息内容，比如，在实际实现时，我们每隔 100 万个时钟周期保存一个 TCB 完整信息，这样可以使得重放时，总是可以在重放点往前寻找到最近的那个 100 万时钟周期所在的位置开始往后重放即可。检测是否需要完整保存 TCB 的代码如图 3-18 所示，具体的保存操作对应代码如图 3-19 所示。

```
96 void OS_tcb_update()
97 {
98     int i,j,p;
99     for(j = 0; j < TCB_COUNT; j++)
100    {
101        if((tcb_fp = fopen("Sys_file/tcb_record.sys","a+"))==NULL)
102        {
103            printf("Cannot open file 'Sys_file/tcb_record.sys'\n");
104            exit(0);
105        }
106        //fprintf(tcb_fp,"%d ",sim_num_insn);
107        fprintf(tcb_fp,"%d ",sim_num_insn-sim_num_insn_cp);
108        sim_num_insn_cp = sim_num_insn;
109
110        char *token = "#2";
111        fprintf(tcb_fp,"%s ",token);
112        for(i=0;i<tcb_size;i++)
113        {
114            mem_access(mem,Read,TCB_REC[j]+i*0x4,&p,4);
115            fprintf(tcb_fp,"0x%x ",SS_SWAPW(p));
116        }
117        fprintf(tcb_fp,"\n");
118        fclose(tcb_fp);
119    }
120 }
```

图3-19 保存完整TCB信息的代码

任务分析重放操作的逻辑流程图如图 3-20 所示。在此图中可以看到，当用户启动重放分析模式后，首先需要判断是否存在重放文件，即在任务调度信息动态记录时生成的状态记录文件。如果不存在的话，则无法进行重放执行，提示用户不存在重放文件，退出重放模式运行。若存在重放文件，任务分析重放模块首先打开此重放文件，并根据用户指定的重放时间区间，从记录文件中读出相应的任务项信息，此后开始逐条进行解析，如果为静态信息记录，则以表格等形式进行静态的状态显示，如果为动态信息记录，则用图形化的方式显示出其随时间变化的情况。

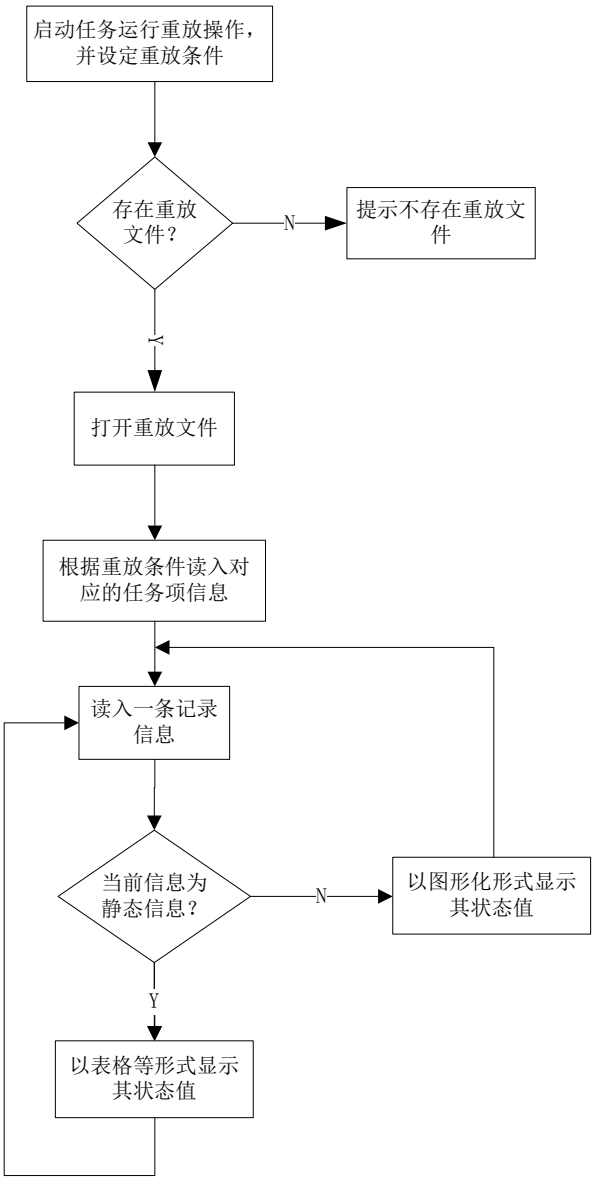

图3-20 重放功能执行流程图

3.4.2 重放配置界面

我们的软件原型系统提供的重放配置界面如图 3-21 所示：

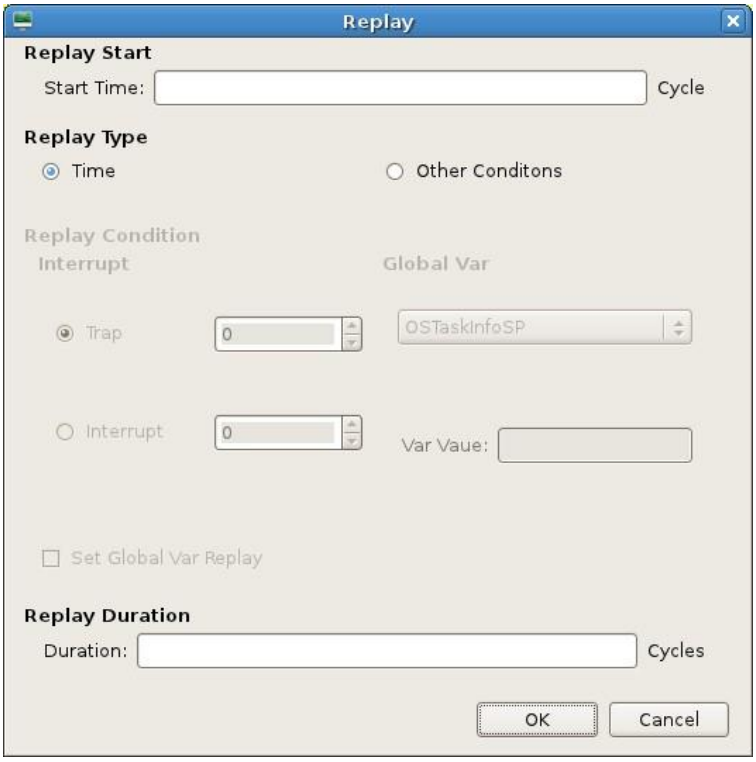

图3-21 重放界面

用户可以选择重放条件，目前提供的重放条件有，根据指定 cycle 长度重放、根据中断类型重放以及根据全局变量重放。现分别介绍一下这三种重放条件设定：

- 系统默认根据指定 cycle 长度重放，用户通过在 Start Time 中输入起始 cycle 点，在 Duration 中输入重放的 cycle 长度，从而可以根据 cycle 范围进行任务重放。
- 选择 Replay Type 为 Other condition，继续选择 trap 或 interrupt 类型，然后输入 cycle 范围，就可以实现在指定 cycle 范围内出现指定中断类型的任务重放。
- 根据操作系统运行的全局变量进行重放。选择 Replay Type 为 Other condition，点击选择框 Set Global Var Replay，用户可以设定指定的全局变量值为重放条件，然后输入重放 cycle 范围，就可以实现在指定 cycle 范围内，该全局变量为预先设定值的任务重放。

图 3-22 为用户设定重放的开始时间和结束时间的情况下，系统执行重放命令后的运行示意图，系统在重放阶段时显示的内容和当时实时显示的内容是一致的。

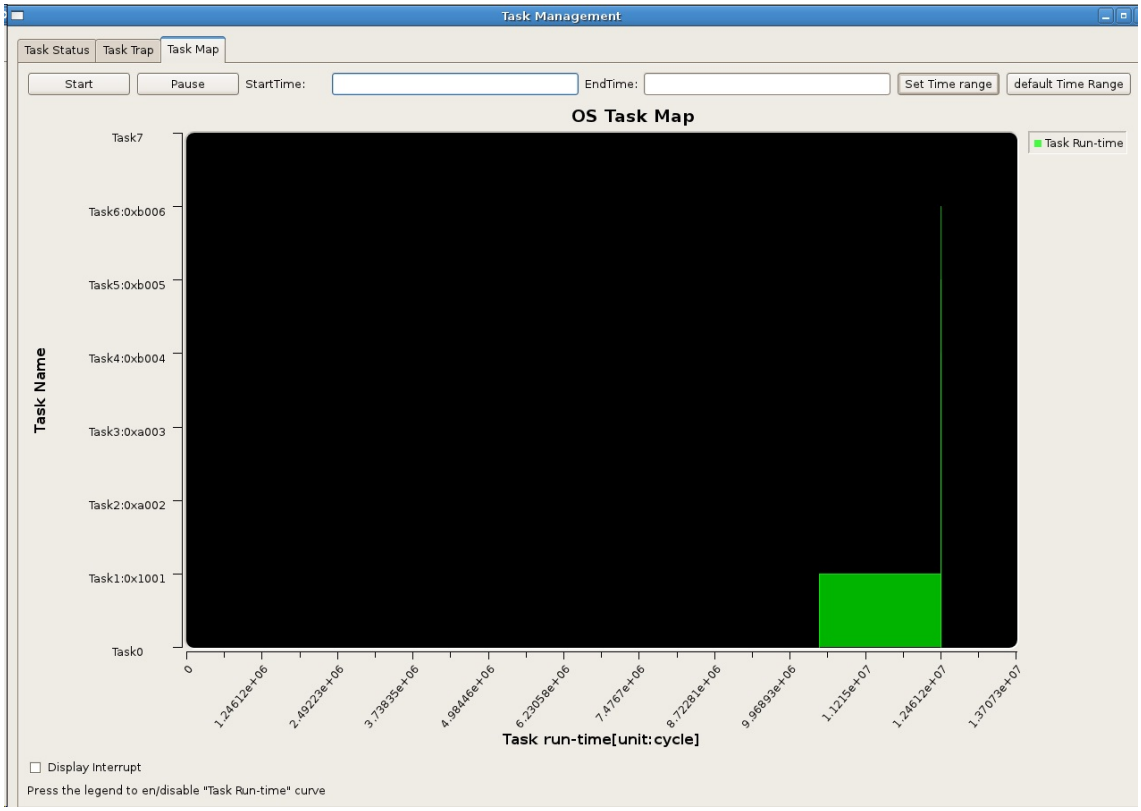

图3-22 重放结果显示示意图

3.5 性能加速

在实现操作系统任务级分析时，我们发现，程序执行的主要时间耗费在对操作系统各个任务控制块变化的记录方面。在操作系统运行过程中，任务的切换比较频繁，产生的任务信息也比较庞大。当我们运行的系统中包含的任务量比较多的情况下，记录的内容会随时间的增加而快速增长，频繁的文件记录操作对时间和空间都是一个极大的挑战。因此，在系统实现时，我们针对实时记录的时间和空间消耗进行了相应的优化。具体的优化策略主要从两个方面进行：优化需要的记录项，以及优化记录的数据大小，下面我们分别针对具体的实现机制进行介绍。

3.5.1 记录格式的优化

为了支持任务信息的实时显示，前台显示的 IDE 界面必须实时地读取任务的状态信息，在真正的系统中，每个时钟周期都有可能有任务信息的刷新，因此，严格来讲，实时的任务显示的刷新频率要求很高，再考虑到每个任务相关的信息很多，比如在我们的例子中，就有 20 多项，在这种情况下，实时显示界面要跟上后台实际硬件执行速度是不可能达到的事情，系统不可能在一个时钟周期的时间

内完成这么复杂的信息记录和读取。即便是在我们的软件模拟平台上，这个依然有很大的难度，因此有必要对实时记录和显示的方法进行一定的优化。

我们发现，虽然在操作系统运行过程中，任务的状态信息可能频繁被改变，但是并不是每次都要改变所有的信息项，实际上，操作系统对任务信息的修改每次往往只是更改个别项，大部分信息在较长的一段时间内都保持不变。针对这一特征，我们可以在实时记录和实时显示时进行优化。

```
213 int Mem_tcbread_check(ss_addr_t addr, int nbytes, int *pp)
214 {
215     int i;
216     int p;
217     for(i=0;i<TCB_COUNT;i++)
218     {
219         if(((addr >= TCB_REC[i])&&(addr <= (TCB_REC[i]+TCB_SIZE))))
220         {
221
222             mem_access(mem,Read,addr,&p,4);
223
224             if(SS_SWAPW(p) == SS_SWAPW(*pp))
225             {
226                 return 0;
227             }
228             else
229             {
230                 return 1;
231             }
232         }
233
234     }
235     return -1;
236 }
```

图3-23 检测重复记录项的代码

具体来讲，在实时记录时，我们只记录任务 20 多项信息中的变化项，而对于不变的项不进行记录，其它项的当前值就是最近的一次记录中反映的值。这样，我们可以有效控制记录文件的大小和记录的时间消耗。具体实现时，我们将每次任务信息的记录操作和已有的任务信息进行比对，如果比对结果显示这次写的值和之前的值一样，那么就不做新的记录，否则才重新记录本次任务信息写操作，核心的检测代码如图 3-23 所示。

在文件中的具体记录格式如下图 3-24 所示，图中列出了针对任务状态项的记录，以及针对中断和全局变量变化情况的记录。

```
1431 10254340 #4 SYSRAMPage 0x32768
1432 10273982 #4 OSIsrMap 0x2048
1433 10274025 #4 OSIntNestType 0x256
1434 10274047 #1 0x1001 0x16 0x9010c3
1435 10274048 #1 0x1001 0x17 0x40003cf0
1436 10274049 #1 0x1001 0x18 0x40003cf4
1437 10274075 #4 OSIsrMap 0x2048
1438 10274106 #4 SYSRAMPage 0x32768
1439 10274116 #1 0x1001 0x12 0x0
1440 10274167 #1 0x1001 0x12 0x1
1441 10274176 #1 0xb004 0x14 0x9
1442 10274213 #4 OSIsrMap 0x2048
1443 10274348 #4 SYSRAMPage 0x32768
1444 10293980 #4 OSIsrMap 0x2048
1445 10294023 #4 OSIntNestType 0x256
1446 10294073 #4 OSIsrMap 0x2048
1447 10294104 #4 SYSRAMPage 0x32768
1448 10294114 #1 0x1001 0x12 0x0
1449 10294165 #1 0x1001 0x12 0x1
1450 10294174 #1 0xb004 0x14 0x8
1451 10294211 #4 OSIsrMap 0x2048
1452 10294346 #4 SYSRAMPage 0x32768
1453 10313982 #4 OSIsrMap 0x2048
1454 10314026 #4 OSIntNestType 0x256
1455 10314048 #1 0x1001 0x17 0x40003cd8
1456 10314049 #1 0x1001 0x18 0x40003cdc
1457 10314075 #4 OSIsrMap 0x2048
1458 10314106 #4 SYSRAMPage 0x32768
1459 10314116 #1 0x1001 0x12 0x0
1460 10314167 #1 0x1001 0x12 0x1
1461 10314176 #1 0xb004 0x14 0x7
```

图3-24 任务信息记录格式示意图

在上述记录机制下，记录文件中只包含当前变化项的值，而对于不变化项的值是没有的，这个可以通过往前查找最近的一个变化项来得到。在实时显示时也需要针对这种机制进行相应的处理，前台 IDE 界面读取出来的内容只包含变化项的信息，因此，IDE 需要记录当前显示的所有任务项信息的旧值，在显示时，将现有旧值和新读取到的变化值拼成完整的任务信息，进行显示。

3.5.2 数据压缩优化

只记录任务信息变化项的方法可以有效避免冗余任务项的记录，从而减少了记录的数据大小，但是当操作系统运行时间很长时，所需的存储开销依然比较大，比如，我们测试的结果发现，在这种机制下，操作系统运行时每分钟大概会产生近 2MB 大小的记录文件，这个空间开销依然比较大，长时间运行之后，需要占用很大的磁盘空间。

针对这一问题，本文进一步采用数据压缩的方法进行优化，我们通过采用高效的压缩算法对待记录任务项信息进行压缩后再进行存储，以进一步减少记录文件大小。

在实现时，我们按照各实时数据的信息特点，分别进行压缩方式的设计。

（1）文本记录数据类型

当监测机制将监测到的数据记录进文本文件时，实数数据就会被转化为字符放入文本进行存储。比如实数 15，如果以二进制格式存储到文本文件，则此数值会占 4byte 的存储空间（1111）；如果以八进制格式存储到文本文件，则占用 2byte 的存储空间（17）；如果以十进制格式存储到文本文件，则占用 2byte 的存储空间（15）；如果使用十六进制格式存储，则只会占用 1byte 的存储空间。由此可见，在同等数值下，使用十六进制格式存储，则占用最少的空间。因此在本文中，实时监测机制的监测记录文件默认使用十六进制的数据表示方式。

（2）时间点信息

时间点信息作为数据记录的时间刻度，它存在于各个记录类型中，用来标记各个记录发生的时间。在 TOS 监测机制中的时间点机制是以指令运行 cycle 数为单位，此时间点监测值具有如下特点：

记录值量级大。首项记录值通常在百万量级以上，长时间运行以后记录值范围巨大，会出现数值溢出的情况。

记录值逐项递增。由于时间的递增性，时间点记录信息维持单调递增的特性。

相邻两记录值之间差值范围有上限。通过对 TOS 运行中的时间点信息进行统计，统计数据显示两个相邻两个时间点记录项之间数值差距在 20000 以下。

针对上述记录信息特点，我们借助差值压缩算法的思想，提出一种相对差值记录的压缩方式。这种方式的设计思想是为防止时间点记录信息溢出并缩短时间点信息占用空间，使用字典记录初始时的时间点信息，接着每有新时间点信息时只记录当前新值与前一时间点信息的差值，对于第一条记录项时间点信息为零。在这种方式下，时间点信息记录项的值范围锁定在 20000 以内。由于 20000 在 214 与 215 之间，因此可以使用一个字中的部分比特来表示时间点记录信息。其具体的差值记录代码如图 3-25 如下：

```
239 void Mem_tcb_record( ss_addr_t addr)
240 {
241     int i;
242     for(i=0;i<TCB_COUNT;i++)
243     {
244         if(((addr >= TCB_REC[i])&&(addr <= (TCB_REC[i]+TCB_SIZE))))
245         {
246             int p,q;
247             FILE * tcb_runstat_fp;
248 
249             char *token;
250             mem_access(mem,Read,TCB_REC[i]+0x4,&p,4);
251             int number = (addr-TCB_REC[i])/0x4 + 1;
252 
253             if(number == STATUS_INFO)
254             {
255                 *token = "#1";
256                 if((tcb_runstat_fp = fopen("Sys_file/tcb_runstat_record.sys","a+"))==NULL)
257                 {
258                     printf("Cannot open file 'Sys_file/tcb_runstat_record.sys'\n");
259                     exit(0);
260                 }
261                 fprintf(tcb_runstat_fp,"%d ",sim_num_insn-sim_num_insn_cp);
262                 sim_num_insn_cp = sim_num_insn;
263                 //fprintf(tcb_runstat_fp,"%d ",sim_num_insn);
264                 fprintf(tcb_runstat_fp,"%s ",token);
265                 fprintf(tcb_runstat_fp,"0x%x ",SS_SWAPW(p));
266                 fprintf(tcb_runstat_fp,"0x%x ",number);
267                 mem_access(mem,Read,addr,&q,4);
268                 fprintf(tcb_runstat_fp,"0x%x ",SS_SWAPW(q));
269                 fprintf(tcb_runstat_fp,"\n");
270                 fclose(tcb_runstat_fp);
271             }
272             else if(number == TRAP_INFO)
273             {
274                 //record trap infomation
275             }
276 
277         }
278     }
279 }
```

图3-25 时间差值记录过程代码

（3）任务运行状态类信息

在任务状态数据中有一类数据，它们的数据取值有固定的取值区间，比如 TOS 中任务状态记录项有 Ready、Delay、Pend、Suspend 和 Run 五种状态，在 TOS 内部分别用 0x01、0x02、0x04、0x08 和 0x10 表示以上各状态；如果按照原始的记录方法，我们应该将从 TOS 内部获取的状态值进行记录，虽然这种方式可以记录准确的信息值，但是冗余现象仍然存在，0x01 这个记录信息在文件中会耗费 4byte 的存储空间。

对于这一类数据，我们提出如下构造方式：首先构造数据字典，然后为每一个字典中的长字符指定对应短字符。经过分析，在所监测的实时数据中，任务名、trap 类型和全局变量名的数据特点也与任务运行状态类相似。

（4）恒值数据串

我们发现，任务实时状态数据中记录的几十基本信息项中有大部分信息在开始运行时赋予初值之后，其在运行中值是不再改变的。在这些为恒值的项中又大

部分项信息恒为 0，而其余项恒为初始指定的非零实数值。针对这些恒为 0 的信息项，每一项在记录文件中会占用 4byte（0x0 ），由于这些项一直为恒值，如果不予记录将会省去大量的空间占用。因此在这里我们可以按如下流程进行处理：统计恒为 0 的项的项号，在监测中若遇到恒为 0 的项则跳过，不予记录。

（5）重复数据串

对于连续重复的数据，我们以字为单位，进行重复次数为 1 的游码压缩法。若一个单位字符串连续重复 2 次或 2 次以上，则使用字符串+出现次数的方式记录；对于重复一次的只记录字符串。

3.6 本章小结

本章我们针对上一章介绍的嵌入式操作系统任务分析机制的三个主要功能进行了原型系统的实现，我们的系统在 Sim697 软件模拟器和前台 IDE 界面的基础上实现了和任务机制相关的功能，我们定义了相应的数据结构用于任务信息的记录，我们在 IDE 界面上完成了任务信息的实时显示以及重放显示。最后，为了减少任务信息记录所需的空间大小，我们提出了一些优化机制。在下一章中，我们将基于本章所介绍的原型系统，针对本文所提出的任务分析机制进行相应的性能评估。

第四章 实验结果及分析

本章基于 Sim697 软件模拟器平台，使用测试程序针对任务分析机制的具体实现方法进行性能评估和分析。本章组织如下，4.1 节介绍测试环境。然后 4.2 节我们分别评估任务分析机制中，数据实时记录性能、实时显示性能、以及重放性能，具体是在第 4.2.1 小节中我们针对只记录变化项的方法进行评估，第 4.2.2 小节评估进一步采用了数据压缩优化后的性能。最后在第 4.3 节中，我们对本章内容进行小结。

4.1 实验评估环境介绍

我们的实验基于 AT697 模拟器软件环境上进行，宿主机配置以及目标系统配置的具体信息如表 4-1 所示：

表4-1 测试环境参数

宿主机配置	CPU	Intel(R) Xeon(R) E7420 CPU, 2.13GHz
	内存	64GB
	OS	Red Hat SMP Linux，内核版本 2.6.18
	编译器	Gcc 4.1.1
目标模拟器配置	模拟目标平台	AT697 处理器，SPARC v8 架构
	IDE 平台	基于 Qt 开发
	目标程序编译环境	SPARC-RTEMS-GCC 4.8
	操作系统	TOS 实时操作系统，最多支持 128 个任务

在实验时，我们在 TOS 上启动不同数目的任务，任务调度机制采用带优先级抢占的调度机制。

4.2 性能评估

4.2.1 记录变化项时的系统性能

在性能分析评估时，我们分别针对记录文件大小、实时显示速度、以及重放速度三个方面进行测试。

（1）记录文件大小

首先，我们在一定的时间内，分别启动不同的任务数，测试得到的记录文件大小的区别。在这里，我们只考虑实时记录的影响，而暂时不考虑其他方面的影

响，因此，在操作系统运行时，我们只加上实时记录功能，而不运行其它功能，比如我们先将前端 IDE 的实时显示功能去掉。

我们将操作系统的任务数分别配置成 2 到 128 个，各自运行一个小时，统计生成的任务记录文件大小，其结果如图 4-1 所示：

图4-1 不同任务数情况下的记录文件大小对比（运行时间：60分钟）

可以看到，随着任务数的增多，记录文件的大小会轻微变大，这主要是由于任务增多之后，任务切换的频率会更高，从而导致任务信息变化相对更加频繁，需要记录的任务项信息也更多。但是由于我们模拟的目标系统只有一个核，任务是轮转运行的，因此，记录信息增加的频率变化并不是非常大，从表中更可以看出，128 个任务的情况下相比 2 个任务的简单情况，记录文件的大小只是增加了 11.8%，总的记录信息文件生成的速率大概在 1.73MB/s 到 1.93MB/s 之间。

总的来说，任务信息的生成速度依然是比较快的，因此我们也采用了一些压缩技术进行优化，在 4.2.2 小节中，我们会针对采用压缩优化后的任务记录机制进行评估。

（2）实时显示性能

系统的实际运行速度除了受到实时记录速度的影响之外，同样还会受到实时显示速度的影响，如果实时显示的速度赶不上系统的实际运行速度，那么也需要让系统运行速度降下来，以保证前端 IDE 界面显示的同步。

这里我们测试一下前端 IDE 界面的实时显示的速度，和之前的方式类似，我们让 TOS 操作系统运行 128 个独立任务，在界面显示时，任务运行状态信息通过

图形化的“任务总运行图”进行显示，而任务的其它细节信息则通过表格数据的形式进行实时更新显示。

为了准确测试实时显示的性能，我们通过一个快速记录生成器实时生成测试所需的任务记录信息，然后针对不同的刷新频率进行测试。由于每次的刷新都有一定的开销，因此不同的刷新频率会影响到实时显示的速度，刷新频率越快，通常显示所需的开销越大，但如果刷新频率太慢又会影响实时显示的效果，在测试时，我们将刷新频率设置成最高每毫秒刷新一次，最低每 1000 毫秒刷新一次，测试得到的结果如图 4-2 所示：

图4-2 不同刷新频率下的显示速度（128任务）

我们希望前台的实时显示速度能和后台模拟器的运行速度比较匹配，这里，我们用每秒能够显示 TOS 运行多少 cycle 数生成的记录数据来衡量显示速度，比如 1M cycle/s 意味着前台每秒钟可以实时显示 TOS 运行 1M 个 cycle 后生成的任务信息。

从图 4-2 中可以看到，随着刷新频率的降低，显示速度逐步加快，当刷新频率大于 100ms 每次时，显示速度已经达到每秒 9M cycle 以上，这个速度已经完全赶上了后台模拟器的执行速度，而且 100ms 刷新一次足够保证前台显示的实时性，用户基本不会感觉到显示的停顿。因此，在我们的系统中，采用 100ms 刷新一次的频率进行处理。

（3）重放速度

任务信息的重放可以在离线状态下进行，不需要实时的刷新，只需要将符合用户指定条件范围内的任务信息依次显示出来即可，在本文中，我们测试了操作系统启动不同任务数，且运行相同时间后（1 小时）生成的记录文件的重放速度，其结果如表 4-5 所示。

图4-3 重放速度对比

可以看到，单纯的重放比实时显示的速度要快很多，这是由于重放避免了反复的刷新操作，测试结果表明，重放速度会随着任务数的增加稍微变慢，但是整体来讲影响并不明显，从 2 个任务到 128 个任务的速度差距在 7%以内，这说明我们的方法具有比较好的适用性，系统可以重放速度比实时刷新速度提高了 20~30 倍，这完全可以满足实际的调试要求。

（4）任务分析机制对速度的影响

从上面的性能分析可以看出，重放功能是离线进行的，对系统的实际运行速度没有影响，而实时记录以及实时显示两个功能是在系统运行过程中实时完成的，会直接影响到系统的实际运行速度。这里我们将系统加上任务信息记录和实时显示功能后的运行速度和不含任务分析机制的系统运行速度进行对比。我们分别运行 2 到 128 个任务，速度对比结果如图 4-4 所示。

图4-4 任务分析机制对系统速度的影响

可以看到，增加任务分析机制后，系统的实际运行速度会有所下降，但是并不是非常明显，降幅从 13%到 17.5%左右，并且随着任务数的增多，速度下降会相对更明显一些，这主要是由于任务数增多之后，任务信息的记录文件会相对大一些，这会导致需要的处理时间相对增长，从而使得执行速度略微下降。

但是，即便如此，20%以内的速度下降也还是可以接受的，增加任务分析机制后，系统的运行速度依然比较快，测试结果显示均在 2.5MIPS 以上，这完全可以胜任通常的系统调试操作。

4.2.2 压缩优化后的系统性能

针对记录信息的压缩优化主要目的是为了减少记录所需的空间大小，但是，记录时的压缩过程，以及在显示时的解压过程都会引入一定的时间开销，在本小节中，我们针对引入压缩优化后的任务记录机制性能进行评估。

（1）记录文件大小

首先，我们对比压缩前后的记录文件大小，我们分别使用压缩前和压缩后的机制运行相同规模的程序，对比生成的记录文件大小，其结果如图 4-5 所示。

图4-5 压缩前后的记录文件大小对比

图 4-6 给出了压缩后的文件大小相比压缩前的比例，可以看到，随着任务数的增多，压缩机制的效果约明显，在 128 个任务的情况下，我们的压缩优化机制可以减少近 70%的数据存储空间，效果非常明显。

图4-6 压缩比结果

（2）实时显示性能

引入压缩机制后，实现显示时需要先进行数据的解压才能显示，因此，对性

能会有一定影响。图 4-7 给出了在不同刷新频率下压缩前后的显示速度对比情况。

图4-7 不同刷新频率下的显示速度（128任务）

可以看到，压缩机制整体上对不同刷新频率下的性能都会有所降低，但是下降比例一般不超过 10%，针对我们最常使用的刷新频率，100 毫秒刷新一次的情况，速度下降不到 8%。

（3）重放速度

类似地，压缩机制也会对重放速度产生一定影响，图 4-8 给出了不同任务数的情况下压缩前后的重放速度对比情况。

图4-8 重放速度对比

可以看到，压缩机制对重放速度的影响是比较均衡的，下降比例一般在10%~15%之间。比如在 128 任务的情况下，使用压缩机制后，记录文件大小可以节省奖金 70%的空间，但是重放时间增加了不到 15%，因此，这个优化依然是比较有价值的。

（4）系统整体运行速度

使用压缩机制后，任务信息的记录和实时显示都会增加一定的时间开销，因此，系统的整体运行速度会变慢，图 4-9 给出了使用压缩机制前后的运行速度对比。

图4-9 压缩前后系统的整体执行速度对比

可以看到，增加压缩机制后，系统的实际运行速度会有所下降，但是下降幅度比较小，从 4.4%到 9.4%左右，随着任务数的增多，压缩机制的效果会更加，同时对性能的影响也相对较大一些，但是差距依然保持在 10%以内。

4.3 本章小结

本章基于软件原型系统对本文提出的操作系统任务分析机制进行了性能评估分析，我们分别测试了实时记录的性能、实时显示的性能、以及任务信息重放的性能，实验结果表明，通过针对数据记录的优化，使得在加入任务分析机制以及压缩优化机制后的系统运行速度并没有下降过多，实时显示的性能也得到了满足，并且，我们的系统的重放速度很快，可以有效满足实际系统调试的需求。

第五章 结 论

在嵌入式系统，特别是实时嵌入式系统中，不仅对逻辑的正确性有着基本的要求，而且对任务的执行时间有着更高的要求。在操作系统中，任务的执行时间会受到任务的调度、资源的竞争、中断等因素的影响。传统的系统级调试无法获取这些信息，而任务级调试可以，它可以提供更高层次的调试功能，能够在操作系统的框架中分析被调试程序的运行行为，从而发现任务级的故障。

5.1 本文的主要贡献

本论文主要针对嵌入式操作系统中的任务级分析机制进行研究，研究如何针对任务的执行信息进行实时记录、实时显示、以及快速重放。本文的主要贡献如下：

- 提出了一套完整的针对嵌入式操作系统执行过程中的任务信息进行分析的机制，包括任务信息的实时记录、实时显示、以及快速重放。实时记录机制可以在操作系统运行过程中实时地将所有任务相关的信息完整记录下来，实时显示机制可以以图形化的方式实时显示出所有任务的当前运行状态，重放机制可以根据用户指定的条件对过去的任务进行快速的重放。
- 基于 AT697 处理器软件模拟平台和实时操作系统实现了一个完整的任务分析系统。该系统包含一个 AT697 软件模拟器，它完整模拟了 AT697 处理器的指令架构以及结构功能，在执行过程中能够监测操作系统对任务信息的修改，并对其进行记录。系统还包括一个前台的 IDE 界面，该界面可以实时显示操作系统运行过程中的任务信息变化情况，并且可以根据之前的记录针对特定的信息进行重放。引入任务分析机制后对系统性能影响最大不超过 17.5%。
- 提出了一种针对实时记录信息进行压缩的机制，以节省实时记录所需的空间开销。我们根据任务信息记录的不同类型分别采用不同的方法进行压缩。实验结果表明，采用压缩机制后的任务分析系统相比压缩前最大可以减少 70%的记录大小，而系统整体执行速度仅下降不到 10%。

5.2 下一步工作的展望

随着技术的进步以及应用的需求，嵌入式系统开始逐渐向多核方向发展，在多核的复杂环境下，对操作系统任务调度实时性的要求将变得更加难以实现。任务调度引起的故障也将变得更加常见，在这种情况下，任务级分析调试机制就显

得更加的重要，必将受到越来越多的重视。本文针对嵌入式实时操作系统中的任务分析机制进行了研究，并实现了一个可用的软件原型系统，但是，本文的工作仍然还有很大值得优化的空间，在后续的工作中，我们准备基于以下方向针对操作系统任务级分析机制进行进一步的优化：

（1）针对实时记录文件采用更加高效的压缩算法，以进一步减少记录所需的文件大小，满足更长时间的程序调试需求。

（2）针对实时显示进行加速优化，在保证刷新频率的情况下研究如何减少刷新所需的开销，以保证当实时记录速度得到大幅度提高的情况下，实时显示依然能够满足其所需的速度要求。

（3）将本文所研究的任务分析机制进行推广应用，包括推广到其它指令架构的系统中，以及推广到多核系统中进行应用。

致 谢

在本论文即将结束之前，首先我要衷心感谢我的导师陈绍刚教授和石媛老师。从论文开题、中期到最后论文的撰写以及论文的修改，论文的最终完成，自始至终都是在他们的精心指导下完成的。论文中的许多观点和内容以及表述都是在他们的指引下得以付诸实现和完善的。

同时，我也要感谢我的同学及同事，英文相关资料的查阅十分困难，他们牺牲自己宝贵的休息时间，给我提供了大量的论文撰写参考材料，保证了论文如期完成，在此，我表示最衷心的感谢！

接着，感谢开发团队中的领导和同事，他们在项目工作过程中给我了很大的支持和帮助，感谢他们帮助我顺利的完成课题的研究、设计、编码、调试以及文档资料整理等工作。在此对他们表示衷心的感谢！

最后，还要再一次感谢陈绍刚教授对我的无私帮助和指导，感谢电子科技大学对我的悉心培养。在此谨向陈老师及电子科技大学致以诚挚的谢意和崇高的敬意，谢谢！

参考文献

[1] C. Haller. The ZEN of BDM[R]. Macraigor Systems Inc, 1997. 3-6

[2] E. Sutter. Monitor-Based Debugging[R]. Embedded.com, 2003. 1-4

[3] H. Agrawal, R. DeMillo, and E. H. Spafford. An execution backtracking approach to debugging[J]. lEEE Software, 1991, 8(3): 21-26

[4] D. Bhatt, A. Ghonami and R. Ramanujan. An instrumented testbed for real-time distributed systems development[C]. In Proceedings of 7th IEEE Real-Time Systems Symposium, 1987. 241-250

[5] P. Corsini and C. A. Prete. Multibug: interactive debugging in distributed systems[J]. IEEE Micro, 1986, 6(3): 26–33

[6] G. Goldszmidt, S. Katz and S. Yemini. High-level language debugging for concurrent programs[J]. ACM Trans. Computer Systems, 1990, 8(4), 311-336

[7] R. McLear, D. Scheibelhut and E. Tammaru. Guidelines for creating a debuggable processor[C]. In Proceedings of 1st Internatianal Conference on Architectural Support for Programming Languages and Operating Systems. ACM SIGPLAN Notices, 1982, 17(4), 100-106

[8] H. Chen, C. Kao and I. Huang. Analysis of Hardware and Software Approaches to Embedded In-Circuit Emulation of Microprocessors[C]. In Proceedings of the seventh Asia-Pacific conference on Computer systems architecture, 2002, 127-133

[9] M. Shobaki. On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems[C]. In Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications, 2002,1-9

[10] S. Howard. A Background Debugging Mode Driver Package for Modular Microcontroller. Semiconductor Application Note AN1230/D[R], Motorola Inc., 1996. 35-47.

[11] IEEE Std. IEEE Standard Test Access Port and Boundary-Scan Architecture[R]. Technical Report. IEEE, 2001, 1532-2001

[12] C. Swingler. Using BDM/JTAG emulators for functional diagnosis and test[R]. Embedded.com, 2002. 1-3

[13] 技术文档. Lambda 2.1 技术白皮书[R]. 北京科技金银北京成功技术有限公司，2002. 7-9

[14] 任昭绪，罗蕾.跨平台嵌入式任务级调试代理的研究与实现[J].福建电脑，2005 年第 3 期，56 页

[15] 张根源.嵌入式系统与 Internet 技术[J].微计算机信息，2000, 16(3):19

[16] F. Halsall and S. Hui. Performance monitoring and evaluation of large embedded systems[J]. Software Engineering Journal, 1987, 2(5): 184-192

[17] 张静.嵌入式软件任务级调试技术研究与工具实现[D] .成都：电子科技大学，2003,9-21

[18] 王庆江，董渭清，张琳等.嵌入式系统及其开发领域典型特性分析[J].计算机应用研究，2002,4,11-13

[19] 邵贝贝.嵌入式实时操作系统 μC/OS-II[M].北京：北京航空航天大学出版社，2007, 1-21

[20] 罗蕾.嵌入式实时操作系统及应用开发[M].北京：北京航空航天大学出版社，2011, 13-16

[21] 侯歆武.嵌入式系统任务级调试器的研究与实现[D].西安：西安电子科技大学， 2012, 11-15

[22] 费训.任务级调试的研究与实现[D].成都：电子科技大学，2005, 21-27

[23] 杜雄.嵌入式系统任务级调试器的研究与实现[D].武汉：华中科技大学，2005, 7-24

[24] 冯伟.嵌入式系统多级调试技术的研究与实现[D].成都：西南交通大学，2008, 1-30

[25] http://qt.digia.com/[EB]

[26] P. Koopman. Embedded system design issues(the Rest of the Story)[C]. In Proceedings of 1996 International Conference on Computer Design. IEEE Computer Society, 1996, 310-317

[27] 范涛，刘高辉，叶笑春等.SPARC 平台模拟器源码级调试系统的研究与实现[J].计算机工程与应用，2013,49（4）, 65-70

[28] 章辉,桥井,高桥,柏琦峰.基于仿真器的源码级调试器设计与实现[J].计算机工程与设计, 2010, 31(8), 1685-1688

[29] 高翔，张福新，汤彦等.基于龙芯 CPU 的多核全系统模拟器 SimOS-Goodson[J] .软件学报，2007. 18(4): 1047-1055

[30] W. Richard Stevens. UNIX 环境高级编程[M].尤晋元等译，北京：机械工业出版社，2002, 132-136

[31] http://qt-project.org/[EB]

[32] 蔡志明，卢传富，李立夏.精通 QT 编程[M].北京：电子工业出版社，第 2 版，2011, 1-635

[33] 陈刚.Eclipse 从入门到精通[M].清华大学出版社，2005, 1-15