基于服务体/执行流模型的 MiniOSEK 操作系统①

王 仁　李 曦　陈香兰　(中国科学技术大学 计算机科学与技术学院 安徽 合肥 230027;

中国科学技术大学 苏州研究院 江苏 苏州 215123)

摘 要： 为解决现有 OSEK 操作系统实时性不高、效率低等问题，提出了一种新的 OSEK 操作系统——MiniOSEK。MiniOSEK 是建立在服务体/执行流模型的基础上，并符合 OSEK/VDX 标准规范的嵌入式操作系统。其设计思想是将消息驱动的思想引入到传统 OSEK 操作系统中，即当一个任务调用 OSEK 操作系统规范规定的应用程序编程接口(API)函数时，只需要向该接口函数发送一个消息。这样可以减少上下文切换所需的时间，提高嵌入式系统的实时性。

关键词： 服务体/执行流模型；OSEK/VDX 规范；嵌入式操作系统；消息驱动；应用程序编程接口

MiniOSEK Operating System Based on Servent/exe-Flow Model

WANG Ren, LI Xi, CHEN Xiang-Lan (Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China; Suhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China)

Abstract: To solve the problems existing in the OSEK operating systems, such as low real-time performance and efficiency, this thesis proposes a new OSEK operating system, namely, MiniOSEK, which is built on the servent/exe-flow model, and in accordance with the OSEK/VDX standard. The design idea of MiniOSEK is introduce message-driven idea into OSEK operating system. When a task needs to call an API function which conform OSEK OS standard, it just needs to send a message to this API function. In this way, the time required for context switch can be shortened and the real-time of the embedded system can be improved.

Keywords: servent/exe-flow model; OSEK/VDX standard; embedded operating system; message-driven; application programming interface

1 引言

为了给汽车应用软件的开发提供统一接口，欧洲汽车工业界提出了 OSEK/VDX 规范体系[1]。其中 OSEK 是指 1993 年德国汽车工业界提出的“汽车电子的开放式系统及接口软件规范”(open system and corresponding interfaces for automotive electronics)，VDX 指的是 1994 年在法国提出的“汽车分布式执行标准”(vehicle distributed executive)。

OSEK/VDX 规范体系包含四个部分，即操作系统

① 基金项目:电子信息产业发展基金(财建[2008]329;工信部运[2008]97)
收稿时间:2010-04-08;收到修改稿时间:2010-05-05

规范(OSEK OS)、通信规范(OSEK COM)、网络管理规范(OSEK NM)和 OSEK 实现语言(OSEK OIL)。

目前国内外关于 OSEK 操作系统[2]的研究主要是基于进程/线程模型的，如 Metrowerks 公司开发的 OSEKturbo OS[3]、WindRiver 公司开发的 OSEKworks[4]、清华大学设计的 OSEK 车用嵌入式实时操作系统[5]、浙江大学设计的 SmartOSEK 操作系统[6]等。

基于进程/线程模型设计的操作系统，不管是宏内核的操作系统还是微内核的操作系统，都存在一些缺点，如线程之间的异步通信导致系统的实时性不高；频繁的上下文切换与现场保存降低了系统的效率等。

服务体/执行流模型[7,8]的结构示意图如图 1 所示。服务体/执行流模型由服务体和执行流两类机制组成，其中服务体指的是完成某一功能的代码和数据的集合；执行流是指 CPU 执行代码而形成的一条连续轨迹，执行流通过小端口进入服务体。系统中有一个核心服务体，用于引导执行流通过小端口进入服务体执行代码并提供一些基础服务。

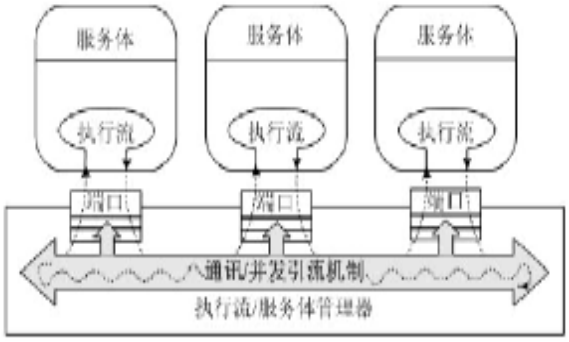

图 1 服务体/执行流模型示意图[7]

服务体/执行流模型在实时性、可扩展性与可维护性等方面具有一定优势，故在该模型的基础上设计一个符合 OSEK/VDX 规范的嵌入式操作系统，即 MiniOSEK，以解决现有 OSEK 操作系统在实时性、效率等方面的不足。

本文的组织如下：第 2 节介绍了基于服务体/执行流模型设计 MiniOSEK 操作系统的设计思想；第 3 节介绍了 MiniOSEK 关键部分的实现及操作系统初始化流程；最后一节介绍了 MiniOSEK 的优缺点以及下一步的研究方向。

2 MiniOSEK的设计思想

MiniOSEK 主要包含两个部分，即核心服务体部分和用户服务体部分。其中核心服务体包含了操作系统运行所需要的一些必备东西，如任务、中断、资源和事件等基本部分的实现，以及一些提供给用户访问的 API 接口；用户服务体包含用户写的代码、数据等，当需要时可以通过发送消息的形式来访问核心服务体提供的 API 函数。在 MiniOSEK 中，用户服务体的概念相当于一个任务，而核心服务体相当于一个提供服务的微内核系统。MiniOSEK 操作系统的运行流程如图 2 所示。

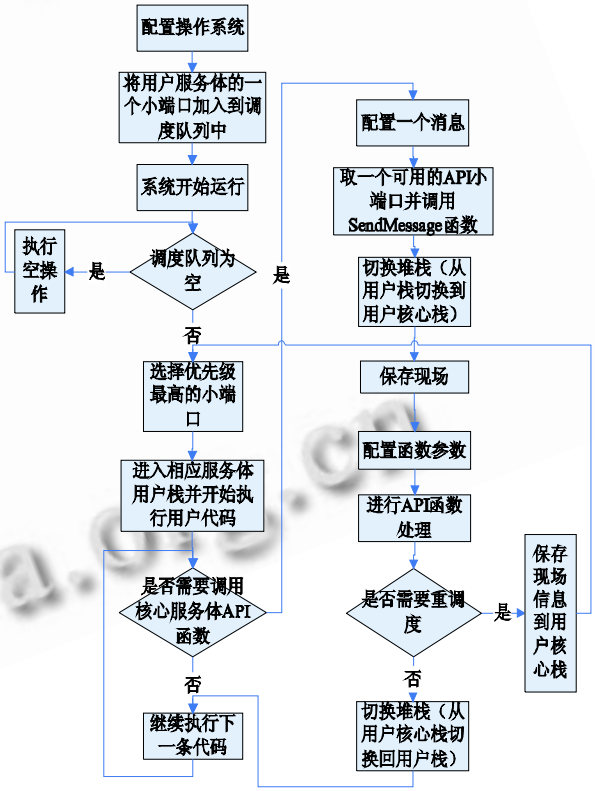

图 2 MiniOSEK 系统运行流程图

MiniOSEK 操作系统的核心思想是消息驱动，即当用户程序需要调用核心服务体提供的 API 函数时，只需向相应 API 的端口发送一个消息。该消息包含了 API 函数的参数和目标端口的编号。然后进行堆栈切换、参数配置、函数执行。执行完之后根据具体函数决定是进行重调度还是返回原用户程序继

续执行。

在 MiniOSEK 中，用户服务体包含两个堆栈，即用户栈和用户核心栈。用户自身的代码在用户栈里运行，当调用 API 时切换到用户核心栈中运行。执行完之后再切换到用户栈。设置两个栈的目的是当一个 API 函数被中断之后，方便上下文的保存和下次执行时的恢复。即将现场保存在用户核心栈中，当下次执行时直接恢复该线程然后继续执行。

3 MiniOSEK的实现

MiniOSEK 实现包括核心服务体的实现和用户服务体的实现两个部分，其中主要是核心服务体的实现。核心服务体的实现包括任务机制的实现、资源管理的实现、事件机制的实现、计数器报警器管理的实现和中断管理的实现。用户服务体的实现包含用户所写代码和数据的封装以及消息的初始化。本节将介绍关键部分的具体实现。

3.1 任务管理

OSEK 规范将任务分为基本任务和扩展任务。基本任务具有 3 种状态：运行状态、就绪状态和挂起状态。扩展任务多了一个等待状态。基本任务只在开始和结束时才有同步点，所以其需要的资源少；扩展任务可以对应不同的时间，在运行中可能有多个同步点，所以对环境要求高。任务激活使用操作系统提供的 ActivateTask 系统服务或者 ChainTask 系统服务。当激活一个任务后，该任务从第一个声明开始执行。操作系统的任务之间的同步通过调度器来实现。OSEK 规范支持 3 种调度方式：完全抢占式调度、非抢占调度和混合抢占调度。

在 MiniOSEK 操作系统中，OSEK 标准中的任务在调度上对应于一个小端口，在存储上对应于一个用户服务体。每个用户服务体在核心服务体中都有注册，并且有一个相应的小端口在核心服务体维护的调度队列中。每次系统调度时，首先找到一个优先级最高的用户服务体，然后由该用户服务体对应在调度队列中的小端口进入相应的用户服务体，执行用户程序和代码。任务是以链表的形式组织的。核心服务体维护了三个队列，即 Ready 队列、Suspend 队列和 Waiting 队列(ECC1 类或 ECC2 类)。

当一个用户服务体加载到系统中之后，首先向核心服务体发送一个消息来注册该用户服务体。然后核心服务体向该服务体的端口发送一个激活消息，并从该用户服务体的空闲小端口队列上取下一个放入核心服务体的 Ready 队列中。如果一个用户服务体被挂起，则将该用户服务体对应的小端口加入 Suspend 队列。如果是扩展类，而且用户服务体的继续执行需要等待某个事件或者某个资源，则将该用户服务体对应的小端口加入 Waiting 队列。

当用户服务体调用核心服务体提供的 API 函数时，需要执行以下步骤：

1) 配置一个消息；

2) 取用户服务体的一个空闲小端口并调用 SendMessage 函数；

3) 进行堆栈切换，从用户栈切换到用户核心栈；

4) 保存现场；

5) 配置所调用的 API 函数参数；

6) 执行相应的 API 函数。

当从 API 函数返回时需要判断此时需不需要进行重调度，如果需要则保存现场到用户核心栈，然后进行重调度；如果不需要则从用户核心栈切换到用户栈，继续执行用户服务体代码。

3.2 资源管理

OSEK 规范规定资源管理要确保：两个任务不能同时占有同一个资源；优先级反转不能产生；运用资源时不能产生死锁；访问资源不能导致进入 waiting 状态；资源管理如果扩展到中断级别，两个任务或者中断例程不能同时占有相同的资源。

为了解决资源管理可能会出现优先级反转或者死锁情况的发生， OSEK 规范中提出了优先级天花板协议。在协议中提到：天花板优先级应该比所有不访问这个资源的任务中最低优先级任务的优先级低，比所有访问这个资源的任务中最高优先级任务的优先级高[1]。从中可以看出一个矛盾，例如：假设有四个任务 T1、T2、T3、T4，T1 和 T2 的优先级分别为 10 和 20，运行过程会占有同一类资源，T3 和 T4 优先级分别为 15，19，不占有资源。按照协议，由 T1、T2 的优先级知道天花板优先级大于 20，由 T3、T4 的优先级知道天花板优先级小于 15，矛盾产生。

为解决矛盾，在 MiniOSEK 中只保证后半句，即资源的优先级遵循一个约定，就是资源的优先级等于所有需要使用该资源的任务的最高优先级加 1。资源

的优先级在系统配置的时候根据任务的占用情况和任务的优先级自动确定。这样就保证了资源不会发生死锁，使任务可以顺利的运行下去，从而满足了 OSEK 规范的要求。

3.3 事件管理

在 OSEK OS 标准中规定，只有扩展任务才有事件。一个独立的事件由它的拥有者和它的名字辨别。

事件管理的核心思想是：用一位表示一个事件，一个任务可以拥有多个事件。设置一个事件，只要设置对应的一位；相应的，清除一个事件，则只需要清除对应的一位。

每个事件可以类似定义为：

```
#define EVENT1      0X0001
#define EVENT2      0X0002
#define EVENT3      0X0004
```

3.4 计数器报警器管理

OSEK 操作系统提供处理循环事件的服务。这些事件比如是一个时钟，它间隔地提供一个中断等。OSEK 操作系统提供两阶段的概念来处理这些事件。循环事件源注册到实现特定的计数器上。基于这些计数器，OSEK 操作系统软件提供报警机制给应用软件。

根据 OSEK 标准，每个计数器必须包含有 AlarmBaseType 中的三项：maxAllowedValue、ticksPerBase、minCycle。maxAllowedValue 规定了计数器可以计数的最大值；ticksPerBase 规定了每个计数器的时间标准，即多少个 ticks 后计数器的计数值加 1；minCycle 规定了循环报警器的最小循环时间。于是，计数器结构设计为：

```
typedef struct Counter
{struct AlarmBaseType base;
 TickType counterValue;
 TickType ticks;
}Counter, *pCounter;
```

参数说明：

1) struct AlarmBaseType base 为：

```
typedef struct AlarmBaseType
{ TickType maxAllowedValue;
   TickType ticksPerBase;
   TickType minCycle;
}AlarmBaseType, *AlarmBaseRefType;
```

2) TickType counterValue：记录计数器的当前值

3) TickType ticks：记录从上次计数器加 1 到现在发生了的时钟中断数，当它达到 ticksPerBase 时，counterValue 加 1，这个值变为 0，重新计数。通过这个值，就可以从一个时钟源得到多个不同时间基准的计数器。

在计数器的基础上，报警器数据结构设计为：

```
typedef struct Alarm
{ unsigned char state;      /* ON - OFF */
  TickType alarmValue;    /* if == counter => alarm */
  unsigned int cycle; /* cyclic inc value */
  struct Counter* ptrCounter; /* reference counter pointer */
  unsigned char TaskID2Activate;
  unsigned int  EventToPost;
  void(*CallBack)(void*);
  void* para;
}Alarm, *pAlarm;
```

参数说明：

1) unsigned char state：表明报警器的工作状态，ON 表示工作，OFF 表示不工作。

2) TickType alarmValue：报警器的报警时间点。

3) unsigned int cycle：如果是循环报警器的话，记录循环报警的周期。

4) struct Counter* ptrCounter：指向引用的计数器。

5) unsigned char TaskID2Activate：要激活的任务 id 或者设置事件的 id(当 EventToPost 不为 0 时)。

6) unsigned int EventToPost：任务要设置事件的掩码。

7) void(*CallBack)(void*)：指向要调用的回调函数，回调函数有一个 void*参数。

8) void *para：指向回调函数的参数

报警器报警的时刻是当报警器结构的 alarmValue 项和引用计数器的 counterValue 值相等。当报警器报警时，可以有三种操作：一是激活一个任务，TaskID2Active 表明要激活任务的 id，此时表示事件掩码的 EventToPost 为 0；二是设置一个任

务的事件，TaskID2Active 表示要设置事件的任务，EventToPost 表示设置事件的掩码；三是调用回调函数，回调函数的入口由 CallBack 指向。

3.5 中断管理

根据 OSEK 标准，OSEK 操作系统包含两类中断服务例程(Interrupt Service Routine)，ISR1 和 ISR2。ISR1 和 ISR2 中断服务例程的区别在于两类中断服务例程过程中可调用的 API 函数不同。ISR1 中断服务例程处理结束后，不进行调度，而 ISR2 中断服务例程结束后，需要进行调度。在中断嵌套的情况下，如 ISR1 被 ISR2 中断，ISR2 处理结束进行任务调度，但此时 ISR1 尚未结束，就会出现不可预料的结果。因此，OSEK 标准中建议，设计时使 ISR1 中断的优先级高于 ISR2，这样就可以避免上述情况的发生。于是在 MiniOSEK 中将 ISR1 中断的优先级设置为高于 ISR2 中断的优先级。

每个中断源在中断允许之前都要被初始化。初始化包含四个步骤：特定模块的初始化，level 指派，允许中断源和设置 SIU 控制寄存器中的中断屏蔽码。

中断现场保护的内容和应用相关。如果中断服务例程 ISR 是由汇编代码实现的，那只需要保存在汇编代码部分使用的寄存器；如果 ISR 是由 C 语言实现的，则需要保存 EABI 规定的所有易失性的寄存器和一些特殊功能寄存器。

一个中断服务例程的处理流程大致如图 3 所示。

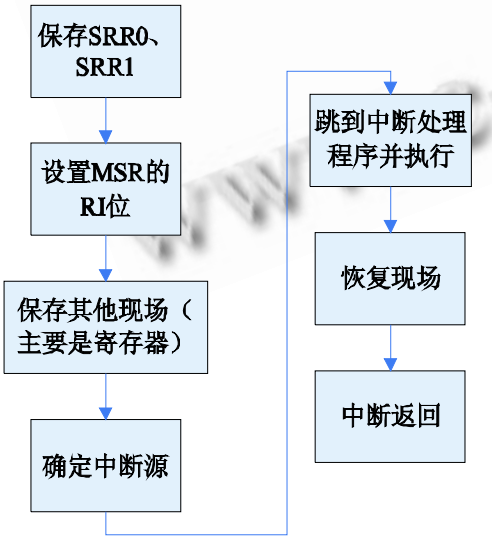

图 3　中断处理流程

可以通过以下的步骤来确定中断源：

1) 检查 SIVEC 寄存器，这个寄存器的值作为一个跳转表的索引；

2) 如果中断源是 level 7，并且应用有中断源对应于 level 7，再检测 UIPEND 寄存器来寻找大于 7 的中断源；

3) 如果有多个相同优先级的中断源，就检测所有相同优先级的中断源；

4) 如果需要，检测是哪一种情况引起的中断。例如，TPU 的 16 个通道是哪一个引起的中断。

3.6 MiniOSEK 开发环境及启动流程

MiniOSEK 采用的是微处理器仿真来达到设计要求，选用的芯片是 MPC5554，选用的仿真器是 QEMU。经过操作系统一致性验证，证明 MiniOSEK 是符合 OSEK OS 标准的，而且在实时性和系统效率等方面比传统的 OSEK OS 要高。

MiniOSEK 的启动主要包括四个部分，即：硬件初始化、软件初始化、任务创建和开始多任务执行环境。

硬件初始化主要是设置微处理器的寄存器，包括 IMMR(内部存储映射寄存器)、MSR(机器状态字寄存器)、BBSMCR(BBS 模块配置寄存器)等。

软件初始化过程主要是在 OSInit()函数中完成的，这个过程需要完成系统全局变量的初始化、任务模块就绪队列的初始化、资源模块资源的注册、事件模块事件的注册、中断模块中断服务例程的注册、时钟模块报警器计数器的注册、建立空闲任务和其它任务同时初始化任务需要的各项内容，最后打开中断进入多任务的运行模式。

用户程序任务的创建和 idle 任务的创建是一样的，但由于用户在提供配置信息的时候可能会出错，故在调用__TaskCreate()之前要对每个参数的合法性进行检查。

在进入 OSStart()函数之后，进入多任务的执行环境。在这个函数中，首先是允许中断，然后在就绪队列中选中最高优先级的就绪任务，并将这个任务从就绪队列中摘下，任务状态修改为 RUNNING，调用 OSStartHighRdy，加载这个最高就绪优先级的执行现场，多任务环境由此建立。

(下转第 5 页)

(上接第28页)

4 结束语

本文分析了传统 OSEK 操作系统存在的一些不足，并在这个基础上提出并实现了一种基于服务体/执行流模型的操作系统——MiniOSEK，分析了其设计思想、关键部分的设计实现以及系统性能分析。

MiniOSEK 是一个基于服务体/执行流模型并且符合 OSEK/VDX 标准的嵌入式操作系统。测试证明 MiniOSEK 是符合 OSEK OS 标准的，并在在实时性、系统效率方面比传统的 OSEK 操作系统有了一定的提高。不足的是在安全性方面 MiniOSEK 还存在一定的问题。故下一步准备在 MiniOSEK 的基础上引入安全机制，即做一个拥有自主访问控制、强制访问控制、审计等的安全嵌入式操作系统。

参考文献

1 黄鹏. 基于OSEK/VDX 的嵌入式车用操作系统研究. 武汉理工大学学报:信息与管理工程版, 2005,27(5): 218－221.

2 OSEK/VDX Operating System Specification Version 2.2.3. The OSEK/VDX Group, 2005.

3 OSEKturbo OS Technical Reference Version 2.2.2. 2003.

4 刘成伟,王尚勇,杨青. 基于 OSEKWorks 开发平台的高压共轨喷油控制系统的研究.内燃机工程, 2004, 25(4):28－31.

5 张宝民,孙晓民.清华 OSEK 车用嵌入式实时操作系统设计.计算机工程与设计, 2004,25(5):661－664.

6 郁利吉. SmartOSEK OS 3.0 的设计与实现[硕士学位论文].杭州:浙江大学, 2007.

7 李宏.基于服务体模型的操作系统设计与实现[博士学位论文]. 合肥:中国科学技术大学, 2004.

8 中国科学技术大学.基于服务体/执行流结构的操作系统. Int Cl: G06F9/44 ZL 200410080994.9, 2007-09-05.