面向逻辑执行时间模型的 Minicore 的内存管理①

刘 杰[1], 陈香兰[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230026)
[2](中国科学技术大学 苏州研究院, 苏州 215123)

摘 要: 随着实时系统在时间关键和安全关键的行业的广泛应用, 程序的时间属性受到越来越广泛的关注. Henzinger 提出了 LET(Logical Execution Time)编程模型, 提供了明确描述时间属性的机制, 确保了系统的时间确定性. 但传统的实时操作系统模型采用了与 LET 截然不同的抽象, 难以很好地支持 LET 编程模型. Minicore 是一种新型操作系统模型, 程序由一组内部没有同步点的服务组成, 具有较好的时间确定性和可控性, 与 LET 编程模型的思想更吻合. 将 LET 的控制模型和 Minicore 的运行模型相结合, 可形成一种具有时间确定性的新型编程框架. 主要描述了该框架的内存管理机制的设计和实现. 文末以智能小车控制系统的实现作为研究实例验证本系统的可行性.

关键词: minicore; Giotto; 内存管理; 逻辑执行时间; 嵌入式

Memory Management of LET-Oriented Minicore

LIU Jie[1], CHEN Xiang-Lan[1,2]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)
[2](Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China)

Abstract: With the wide application of real-time systems in time-critical and safety-critical areas, the timing property of a program is getting more and more attention. Henzinger has come up with LET (Logical Execution Time) programming model, which introduces semantic to deal with timing in an explicit way, assuring the timing predictability of a system. However, traditional RTOS is based on exactly different abstractions and can hardly support LET programming model in a good way. Minicore is a novel operating system model, in which a program is composed of a set of services without internal synchronization point. Minicore is based on a similar abstraction with LET model and minicore program is more time-predictable and controllable. Combining the control model of LET and execution model of Minicore, we can get a novel time-determinable programming framework. This paper mainly describes the design and implementation of the memory management of this framework. An implementation of intelligent car system is presented as a case study to prove the feasibility of this system.

Key words: minicore; Giotto; memory management; logical execution time; embedded

1 引言

随着嵌入式软件系统在汽车电子、航空航天和医疗设备等安全关键和时间关键行业的广泛应用, 软件的时间安全性受到越来越广泛的关注[1]. 传统的 BET(Bounded Execution Time)编程模型最大的缺点在于不具有时间确定性[2], 任务每次执行都可能表现出不同的时间属性. 用 BET 模型实现时间关键的嵌入式控制系统, 不仅成本高、难度大, 且难以通过反复实验或者理论分析证明其时间安全性.

LET[3](Logical Execution Time) 编程模型是 Henzinger 提出的一种能直接描述任务的时间属性的高级编程模型, 该模型认为任务的计算过程消耗一段

① 基金项目:国家自然科学基金(61379040,61272131);江苏省自然科学基金(SBK2012194)
收稿时间:2015-05-05;收到修改稿时间:2015-05-28

固定的逻辑时间——由编程者根据系统需求指定(逻辑时间≥WCET). 逻辑执行时间不受实际运行时间的影响, 所以 LET 编程模型具有较好的时间确定性.

近年来, 设计和实现具有时间确定性的编程框架或者操作系统成为了研究热门[4-8], 基于 LET 编程模型的相关研究是其中一个重要的分支.

现有的基于 LET 的编程框架均建立在传统的进程/线程模型的操作系统之上. 由于传统的进程/线程模型与 LET 编程模型有着完全不同的抽象, 且任务之间存在不确定的同步和竞争[9], 任务的可控性较差, 使得任务的运行难以完全遵循 LET 模型的控制逻辑, 无法取得较好的时间确定性.

Minicore[10]是一种简单的操作系统模型. 传统操作系统中复杂的任务, 在 Minicore 中被拆分成若干简单的服务, 服务内部没有同步点, 各个服务相互协作, 不存在不确定的交互和竞争, 具有良好的时间确定性和可控性.

基于 LET 编程模型和 Minicore 操作系统模型, 可设计出一种具有较好的时间确定性的编程框架. 实现该编程框架涉及多个方面的工作, 本文完成了其中最重要工作之一——内存管理机制的设计和实现.

2　Minicore概述

在 Minicore 中, 一个程序由若干个服务(Service)组成, 服务是调度的单位, 由内部没有同步点的代码和相关数据组成. 系统的执行能力被抽象为执行流(Exe-flow), 执行流按照一定的顺序流经各个服务, 不会被阻塞或者中断.

图 1 显示了 Minicore OS 的架构. 除时钟、内存管理和调度器等基本功能之外, 内核还提供了 Service Rgister、Naming Service 和 Data Driver 作为程序的运行支持. 其中 Service Rigister 用于在系统初始化时将用户设计的服务注册到系统中(即为服务分配存储空间, 建立映射等); Naming Service 建立了从服务名到对应函数入口地址的映射. 在运行过程中, 系统可以利用 Naming Service, 根据请求者指定的服务名称查询对应的函数入口地址; Data Driver 是内核提供的一种数据交换机制, 服务间的数据交换均通过调用 Data Driver 实现.

在原有的 minicore 系统中, 服务直接运行在操作系统之上, Minicore OS 内核负责管理服务的创建、调度运行、和销毁. 为达到更好的时间确定性, E machine[3,7]被移植到 Minicore 中. E machine 是用于解释执行 E code 的虚拟机. 在 Minicore 中, E machine 被作为中间层运行在 Minicore 系统内核和用户服务之间, 它利用操作系统提供的时钟、内存管理、调度器、Data Driver 和 Naming Service 等机制管理服务的创建、调度运行、交互和销毁.

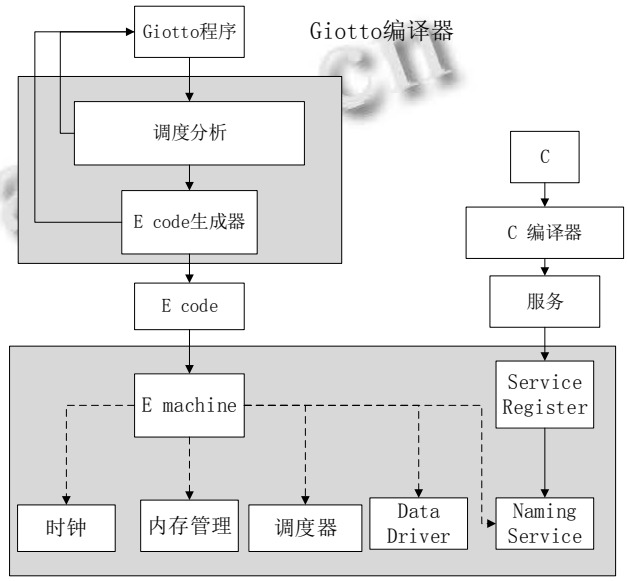


图 1　Minicore OS 架构图

新型的 Minicore 程序由 C 语言和 Giotto 语言共同实现. 其中, C 语言用于实现程序的功能, 编译之后形成服务, 注册到系统中; Giotto 语言用于实现程序的控制逻辑, Giotto 代码将被编译成 E code, 运行在 E machine 之上, 用于控制各个服务的运行.

3　内存管理机制的设计

本文为新型的编程框架设计了全新的内存管理机制, 设计内容包含物理内存组织、地址空间管理和用户接口定义等.

3.1 物理内存布局

静态内存分配机制具有更好的实时性和时间确定性. 动态内存分配机制具有更好的灵活性, 且内存利用率较高. 实时操作系统的内存管理设计需要在两者之间做出平衡的选择, 如 μC/OS-II, RTEMS 和 VxWorks[11].

本系统采用了将动态内存分配机制和静态内存分配机制相结合的内存分配机制, 根据需求预留一部分

物理内, 将这部分内存初始化成常用的多种数据结构, 用作静态内存分配, 当预留的内存不足时再从未预留的物理内存中分配. 这样既能保证一定的时间确定性, 又具有一定的灵活性.

本系统把物理内存划分为三个部分: 静态内存管理区(STATIC ZONE)、KMALLOC 内存管理区和堆内存管理区(HEAP ZONE), 如图 2 所示.

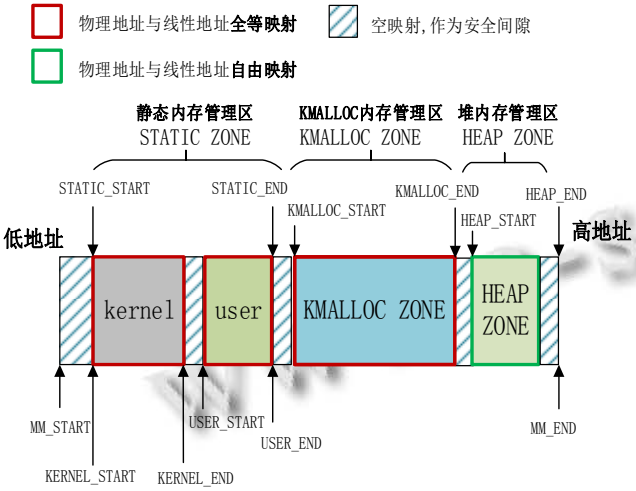


图 2 Minicore 整体内存布局

静态内存管理区 STATIC ZONE.

静态内存管理区(SATATIC ZONE)用于加载系统镜像, 分为 kernel 和 user 两部分.

Kernel 部分用于存储内核代码和数据, 分为.stack, .text 和.data 三个段, 如图 3 所示. 其中.stack 段是内核栈, 是为内核路径的执行或者中断处理函数的执行提供的栈; .text 是编译后得到的内核代码; .data 段包含内核使用的各种数据结构、变量等, 同时也包含预分配的静态对象(Objects).

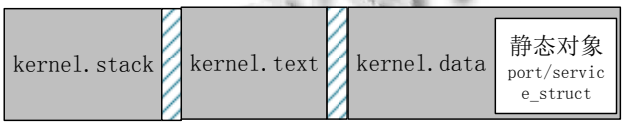


图 3 kernel 部分内存布局

静态对象是指在内核编译时为某些常用结构预留的内存, 如服务控制块 service_struct. 通过预留静态对象可以使内存分配有较好的时间确定性, 缺点是可能会造成内存资源的浪费.

User 部分用于存储用户服务的代码和数据(可能包含用户栈), 如图 4 所示. .text 和.data 段分别用于存储静态编译得到的服务代码和数据, .stack 是服务运行时使用的栈.

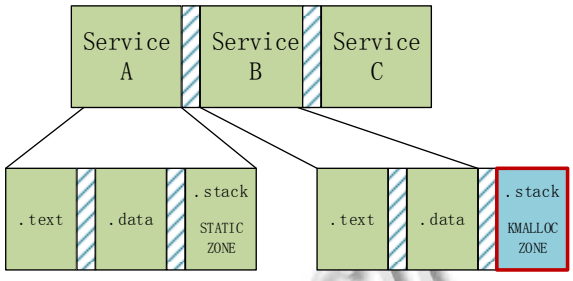


图 4 user 部分内存布局

静态内存管理区中的服务是静态的, 需要通过调用 service_create 函数创建一个可调度的服务实例. 每个服务实例运行所需的栈大小不尽相同, 如果为每个服务分配等大小的栈会造成内存资源的浪费. 本系统采用了一种灵活的设计方案: 将栈的大小交给编写用户服务程序的编程人员指定. 本系统在 service_create 函数中增加了 stack 参数, 作为用户为服务运行实例指定栈的用户接口. 改写后的 service_create 函数包含两个参数, 分别对应的函数入口地址(entry)和为之分配的栈(stack)内存, 定义如下:

service _create (void *entry, void *stack);

为了满足在设计程序时静态指定栈内存, 以及在程序运行时动态指定栈内存两种需求, 本系统还设计了两种分配栈内存的方式: 将栈申明为一个静态数组, 或者调用 kmalloc 动态分配, 如下:

```
int stack1[8192];
service_create((void*)ServiceA, (void*) stack1);
int *stack2 = (int*) kmalloc(sizeof(int)* 2048);
service_create((void*) ServiceA, (void*) stack2);
```

如果栈是一个静态数组, 则将在编译后同服务的.text 和.data 段存储在静态内存区的 user 区间中; 如果调用 kmalloc 进行分配的, 则位于 KMALLOC 内存管理区中.

KMALLOC 内存管理区.

本系统将静态内存管理区之外的物理内存用于实现动态内存分配. 根据用途将这部分内存划分为两部分: 一部分用于满足用户服务对堆内存的请求, 称为堆内存管理区(HEAP ZONE); 另一部分用于满足其他的动态内存需求, 称为 KMALLOC 内存管理区(KMALLOC ZONE).

嵌入式实时操作系统对实时性要求比较高，内存的管理结构在满足功能需求的基础上应当尽可能地简单．因此本文采用简单的“分区内存管理”机制对 KMALLOC 区进行管理．将内存划分为若干个分区(Partition)，每个分区分成等大小的块(Block)．不同分区中块的大小不同，起始地址按块大小对齐，如图 5 所示．

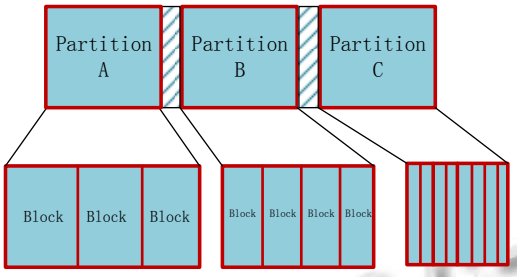


图 5　KMALLOC 内存管理区布局

每个内存块(block)对应一个块描述符(block_struct)．一次同时分配的若干块称为一个区域(region)，如图 6 所示．把每个区域中的第一个块(block)作为该区域的头块(header block)，用其块描述符中的 size 成员指定本区域的大小．

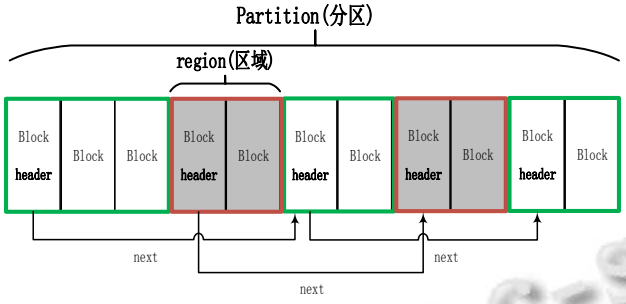


图 6　KMALLOC 内存管理区的组织

块描述符的结构定义如下:

```
typedef struct{
    INT32U size;
    block_struct *header;
    service _struct *owner;
    shr_ctrl_struct *shr_ctrl_blk;
    block_struct *next;
    block_struct *prev;
    block_struct *next_cntxt;
    block_struct *prev_cntxt;
}block_struct
```

初始的时候将每个分区(Partition)中第一个块指定为头块，将它的描述符中 siz 被指定为分区的大小，同时标记为空闲内存区域，将所有块的 header 成员都指向第一个块．

本系统还为每个分区指定了一个分区描述符(partition_struct)，其定义如下:

```
typedefstruct{
    INT32U   blk_size;
    INT32U   nblks;
    INT32U   nfree;
    void     *start;
    block_struct     *free_list;
    block_struct *classified_free_list[SIZE_GROUP_N];
    block_struct*allocated;
    shr_ctrl_struct *shr_mm;
} partition_struct;
```

本系统将每个分区中所有已分配的内存(分为已共享和未共享)和空闲内存分别组织成不同的链表，链表中的元素按照块的地址进行排序，如图 7 所示．链表头为分区描述符 partition_struct 中的 allocated、free_list 和 shr_mm．由块描述符(block)中的 next 指针指向下一个链表元素．

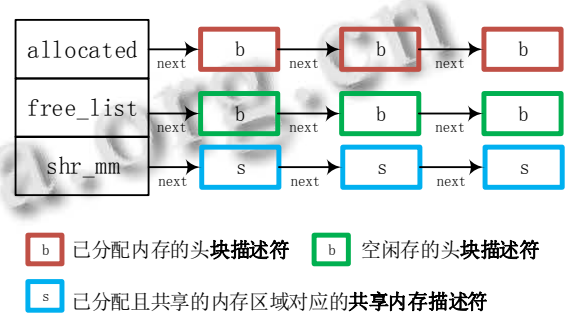


图 7　KMALLOC 内存区组织结构

堆内存管理区 HEAP ZONE.

本系统把堆内存管理区用于满足用户对连续内存的请求，用户程序使用 C 语言中的 malloc 函数获得的内存便是从堆内存管理区中分配的．

本系统根据配置将 KMALLOC 内存管理区划分为若干个分区(Partition)，把每个分区划分为等大小的块(Block)．内存的分配以块为单位，且各块必须连续．

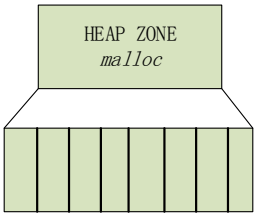


图 8 堆内存管理区结构

和 KMALLOC 内存管理区的区别是: KMALLOC 内存管理区可以被划分为多个分区, 堆内存管理区只包含一分区. KMALLOC 内存管理区中的内存的物理地址和线性地址是相等的, 而 HEAP 内存管理区中线性地址和物理地址是根据实际使用情况任意对应的. 为了记录线性地址和物理内存的对应关系, HEAP 内存管理区使用特殊的块描述符 heap_block_struct, 其定义如下:

```
typedefstruct{
    INT32U size;
    blcok_struct *_header;
    void *virt_addr_start;
    service _struct *owner;
    heap_struct *next;
    heap_struct *prev;
    heap_struct *next_cntxt;
    heap_struct *prev_cntxt;
    } heap_block_struct;
```

3.2 地址映射

为了实现内存隔离, 本设计为各个服务指定了独立的页表, 也为内核指定了一个单独的页表. 将虚拟内存划分为“低地址空间”和“高地址空间”.

“低地址空间”指图 2 中从 MM_START 到 HEAP_START 之间的部分对应的虚存地址空间. 低地址空间部分像其他嵌入式实时系统一样采用单地址空间. 目的是为了减小系统的复杂度, 省去用于维护线性地址与物理地址之间复杂映射关系的结构, 使服务之间的内存共享更简便高效, 保证实时性和时间确定性. 单地址空间还有一个好处就是便于数据共享和函数调用, 同时避免了地址空间切换带来的复杂性.

“高地址空间”指的是图中 HEAP_START 到 MM_END 之间的部分. 多数嵌入式实时系统都采用单地址空间, 本系统主体也是采用单地址空间, 只有堆内存部分采用了多地址空间的机制. 这样既能允许各个用户服务的堆和栈地址空间连续增长, 又避免了物理内存的浪费.

4 接口定义

本系统内存管理机制为用户提供了便捷的用户接口用于实现内存的申请、释放、共享和取消共享等操作, 如表 1 所示.

表 1 用户接口

heap_info_struct **usr_heap_get**(INT32U size, void *err)	申请堆内存
void **usr_heap_put**(void *virt_addr, void *err)	释放申请的堆内存
shr_info_struct **usr_shm_get**(INT32U id, INT32U size, void *err)	获取共享内存块
void **usr_shm_put**(INT32U id, FLAG flags, void *err)	释放共享内存
shr_info_struct **usr_shm_attach**(INT32U id, void *err)	映射共享内存
void **usr_shm_detach**(INT32U id, void *err)	取消共享内存映射
void **usr_grant**(INT32U id, INT32U service _id, void *err)	将共享内存授予给其他服务
shr_info_struct **usr_shm_info**(INT32U id, void *err)	获取共享内存的信息

5 实验和分析

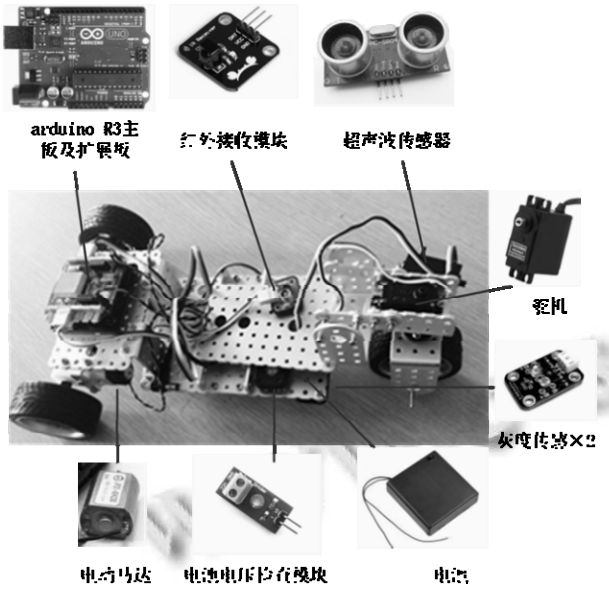
为了验证本系统的可行性, 本文在 Minicore 上实现了一个可自动循迹(黑色轨迹)的智能小车控制系统, 如图 9 所示.

智能小车控制系统包含 1 个处理器,2 个用于循迹的灰度传感器; 1 个用于监测障碍物的超声波传感器; 1 个用于测量电池电压的电压传感器; 1 个用于接收远程遥控信号的红外接收器; 1 个用于控制小车转向的舵机和 1 个提供前进和后退动力的电动马达.

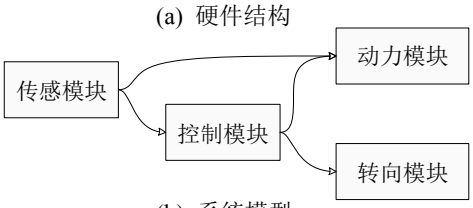
智能小车控制系统由 4 个模块组成: 传感模块、控制模块、动力模块和转向模块(如图 9(b)). 传感模块周期性地对传感器采样, 控制模块根据采样数据控制小车行驶, 动力模块根据控制模块输出的控制参数控制小车前进、后退或停止, 转向模块根据控制模块输出的控制参数控制小车左转、右转或者直行.

实验过程中, 该小车控制系统能正确地控制小车沿预设的黑色轨迹行驶. 在其行驶过程中, 本实验用 80MHz(即精度为 0.0125μs)的逻辑分析仪测量了每次

存储相关的操作所消耗的时间, 其统计结果如表 2 所示. 文献[12]提出的时间确定性的量化模型为: 时间确定性=1-最好执行时间/最坏执行时间. 则实验测得的内存管理相关操作的时间确定性均在 90%以上.



(a) 硬件结构



(b) 系统模型

图 9　智能小车控制系统

表 2　实验过程中相关内存操作的时间统计

	平均时间 (μs)	最小时间 (μs)	最大时间 (μs)	最大偏差百分比(%)
获取静态对象	12.4875	12.35	12.525	1.401401401
释放静态对象	10.6125	10.575	10.7	1.177856302
获取内存块	23.6875	23.225	23.9875	3.218997361
释放内存块	17.4625	17.375	17.6	1.288475304
创建共享内存	31.9375	30.95	33.325	7.436399217
释放共享内存	22.0875	21.75	22.325	2.6032824
获取堆内存	21.7375	21.3375	22.1875	3.910293272
释放堆内存	16.825	16.6125	17.15	3.194650817
映射共享内存	13.85	13.2625	14.2625	7.220216606

6　结语

设计和实现具有时间确定性的编程框架是一个重要的研究课题, 本文设计和实现了一种新型的时间可预测的编程框架的内存管理机制, 是实现该编程框架中重要的工作之一. 本文还以智能小车控制系统为实例验证了本设计的可行性. 实验过程中测得的数据在一定程度上反映本系统的内存管理机制具较好的时间确定性.

参考文献

1 Kirner R, Puschner P. Time-predictable computing. Software Technologies for Embedded and Ubiquitous Systems. Springer Berlin Heidelberg, 2010: 23–34.

2 Hahn S, Reineke J, Wilhelm R. Towards compositionality in execution time analysis-definition and challenges. 6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems. 2013.

3 Kirsch CM, Sokolova A. The logical execution time paradigm. Advances in Real-Time Systems. Springer Berlin Heidelberg, 2012: 103–120.

4 Wang Z, Pu G, Li J, et al. A novel requirement analysis approach for periodic control systems. Frontiers of Computer Science, 2013, 7(2): 214–235.

5 Ayestaran I, Nicolas C F, Perez J, et al. Modeling logical execution time based safety-critical embedded systems in System. 2014 3rd Mediterranean Conference on Embedded Computing (MECO). IEEE. 2014. 77–80.

6 Kloda T, d'Ausbourg B, Santinelli L. EDF schedulability analysis for an extended Timing Definition Language. 2014 9th IEEE International Symposium on Industrial Embedded Systems (SIES). IEEE. 2014. 30–40.

7 Ayestaran I, Nicolas CF, Perez J, et al. A novel modeling framework for time-triggered safety-critical embedded systems. Proc. of the Forum on Specification. Design Languages. 2014.

8 Lee EA. From ptides to ptidyOS, designing distributed real-time embedded systems. Dissertations & Theses-Gradworks, 2011.

9 Hahn S, Reineke J, Wilhelm R. Towards compositionality in execution time analysis-definition and challenges. 6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems. 2013.

10 龚育昌,张晔,李曦,陈香兰.一种新型的构件化操作系统的内核设计.小型微型计算机系统,2009, 30(1).

11 孙琳,刘志丹.嵌入式操作系统内存管理分析与探讨.科技信息,2012,(24):278–279.

12 Grund D. Towards a formal definition of timing predictability. Workshop on Reconciling Performance with Predictability, Grenoble, France (October 2009). 2009.