v 1.1
20221106

嵌入式系统和物联网

信息工程与科学 - UNIBO
a.a 2022/2023

讲师。Alessandro Ricci教授

[模块-2.3] 以

任务为基础的架构

纲要

- 基于任务的架构
  - 与同步FSM的整合
- 任务的合作性调度
- 执行力分析
  - CPU利用率，最坏情况下的执行时间。
  - 问题
    - 超限，抖动
- 有最后期限的任务和基于优先级的调度
  - 静态和动态优先权
- 基于事件触发的FSM的任务

基于任务的架构

- 问题：复杂的嵌入式软件建模和设计--需要适当的方法来分解/

  模块化。

  行为和功能
- 方法。基于任务的架构
  - 嵌入式软件的行为被分解为一个

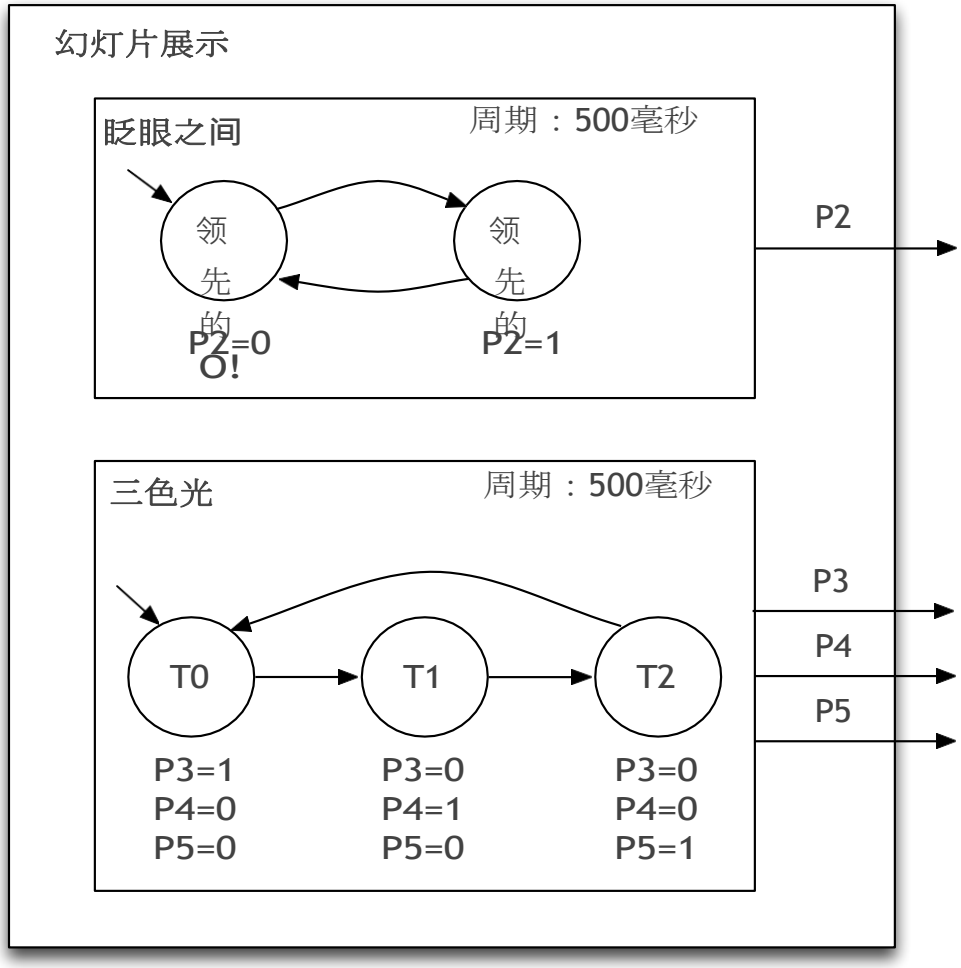
    一组**并发的任务**
  - 每项任务都代表着一个具体的、明确的工作单位/要完成的工作。
  - *每个*任务的行为都可以用FSM来描述
  - 全局行为是并发的FSM执行和互动的结果

举例说明。LED-SHOW

- LedShow的例子。3+1个LED（第68页，[PES]）。
  - 一个LED：闪烁的周期。500毫秒
  - 其他三个LED：依次打开/关闭，时间间隔为500ms。
- 这可以被建模为一个单一的任务/FSM，然而，如果我们将其建模为2个任务/FSM的明确组成，则建模会更容易和简单。
  - 单一任务/FSM版本：更多的状态和转换的数量

框图

- 将每个任务表示为一个块（矩形），显示每个任务/块的输入和输出。



任务分解：优势

- **模块化**
  - 每个任务都是一个独立的模块
  - 在这种情况下，模块的 "接口 "是任务所使用的输入/输出变量/对象的集合（作为输入/输出）。
    - 这些可以与其他模块共享
- 优势
  - 单一任务与整体行为的复杂度降低
  - 调试更容易
  - 可重用性

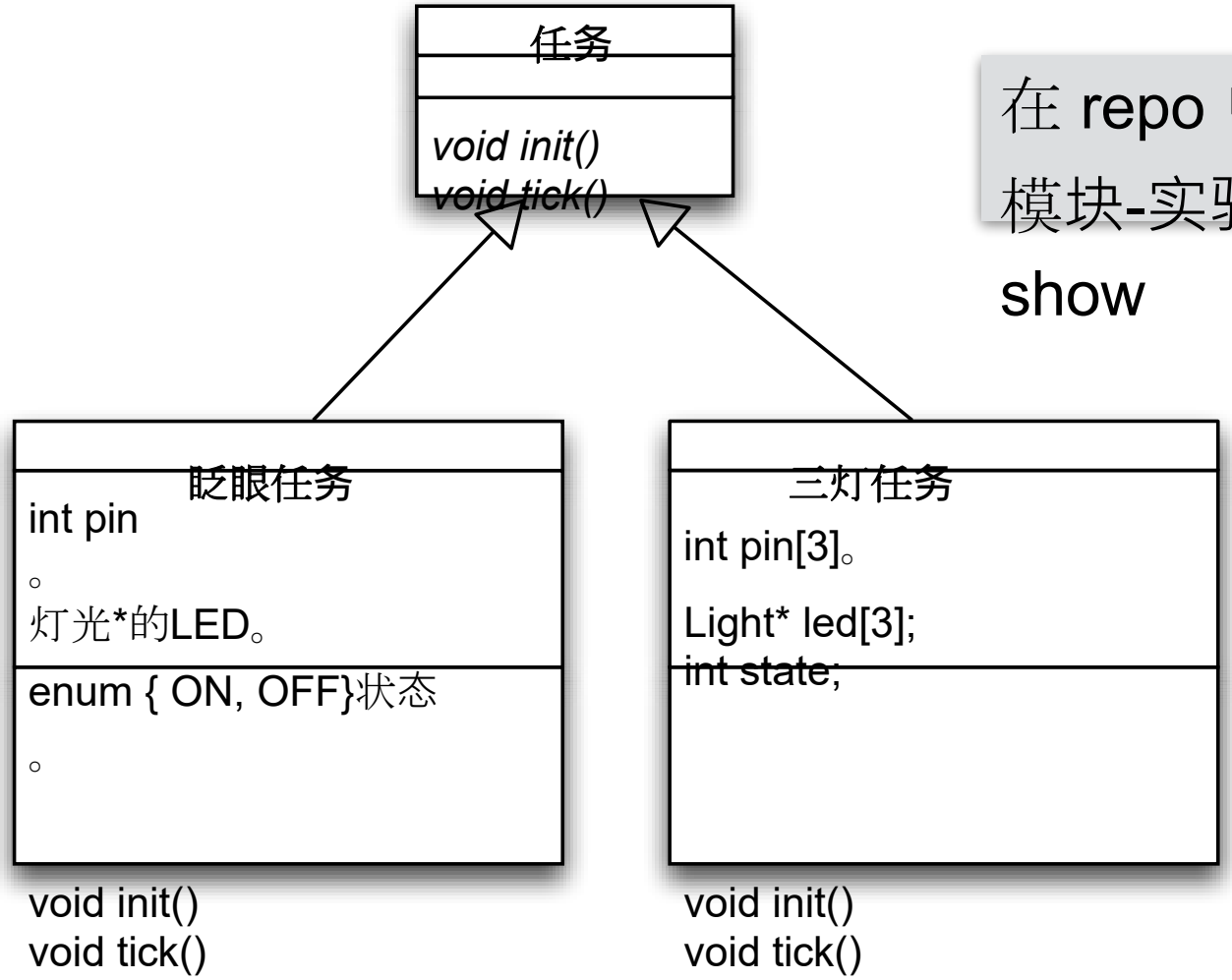
任务分解：挑战

- 任务是**并发的**
  - 它们的执行在时间上是重叠的
  - 在概念上，每个任务都有自己的逻辑控制流程
- 任务可能有*依赖性*，在任务之间产生相互作用，需要适当管理。
  - 互动和协调机制
  - 在我们的案例中，通常是基于共享变量/对象的。

实施基于任务的架构--首先

- 引入一个抽象的基类**任务**
  - **启动**方法
    - 来初始化任务，一旦调用
  - 勾选法（相当于FSM中的 "步骤 "法）。
    - 囊括了任务的行为
    - 定期调用
- 每个具体的任务都是通过扩展这个类来实现的
- 通过定期调用主循环中的tick方法进行任务执行。

Arduino中的LED显示示例



在 repo 中。

模块-实验室-2.3/led-show

Arduino中的led-show例子：任务抽象类

```
#ifndef TASK
#define TASK

类任务 {

公虚空          init() = 0;
  虚空          tick() = 0;



};

#endif
```

arduino中的led-show例子：闪烁任务

```
#ifndef BLINKTASK
#define BLINKTASK

#include
"Task.h"#include "Led.h"

class BlinkTask: public Task {

  int pin;
  灯光*的LED。
  enum { ON, OFF} state;

public:

  BlinkTask(int pin);
  void init();
  空白的tick()。
};

#endif
```

BlinkTask.h

```
#include "BlinkTask.h"

BlinkTask::BlinkTask(int pin){
  this->pin = pin;
}

void BlinkTask::init(){
  led = new Led(pin);
  state = OFF;
}

void BlinkTask::tick(){
  switch (state){
    案关。
      led->switchOn();
      state = ON;
      break;
    case ON:
      led->switchOff();
      state = OFF;
      break;
  }
}
```

BlinkTask.cpp

Arduino中的LED显示示例：三个LED任务

```
#ifndef THREELEDSTASK
#define THREELEDSTASK

#include
"Task.h"#include "Led.h"

class ThreeLedsTask: public Task {

  int pin[3];
  Light* led[3];
  int state;

公众。

  ThreeLedsTask(int pin0, int pin1,
                int pin2) 。
  void init();
  void tick()。
};

#endif
```

ThreeLedsTask.h

```
#include "ThreeLedsTask.h"

ThreeLedsTask::ThreeLedsTask(int pin0, int pin1,
                             int pin2){
  this->pin[0] = pin0;
  this->pin[1] = pin1;
  this->pin[2]  =  pin2
  。
}

void ThreeLedsTask::init(){
  for (int i = 0; i < 3; i++){
    led[i] = new Led(pin[i])。
  }
  状态=0。
}

void ThreeLedsTask::tick(){
  led[state]->switchOff();
  state = (state + 1) % 3;
  led[state]->switchOn() 。
}
```

ThreeLedsTask.cpp

arduino中的led-show示例：主循环

```
#include "Timer.h"#include
"BlinkTask.h"#include
"ThreeLedsTask.h"

定时器定时器。

BlinkTask blinkTask(2);
ThreeLedsTask threeLedsTask(3,4,5);

void setup(){
  blinkTask.init();
  threeLedsTask.init();
  timer.setupPeriod(500);
}

void loop(){
  timer.waitForNextTick();
  blinkTask.tick();
  threeLedsTask.tick()。
}
```

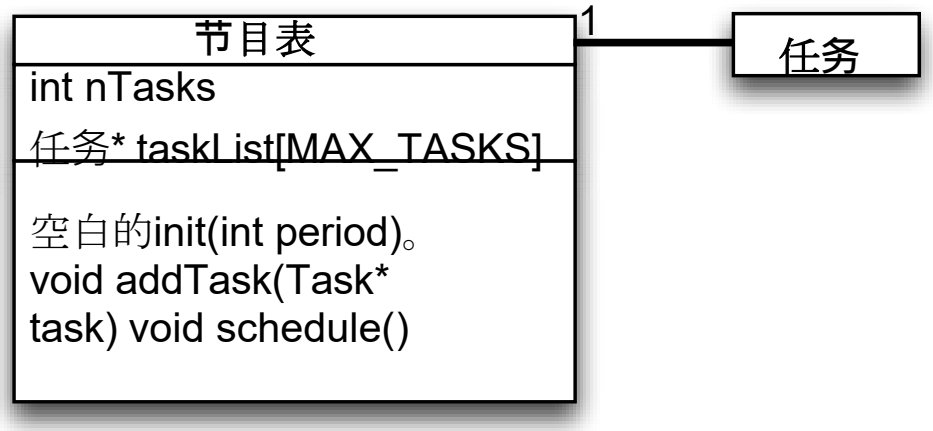
led-show.ino

管理不同时期

- 只要我们想管理不同时期的任务/FSM，就必须这样做。
  - 以跟踪每项任务的具体期限
  - 实现任务调度，使每个任务都有自己的周期被调用。
- 为此，我们实现了一个简单的多任务合作调度器

简单的合作调度器

- 节目表
  - 追踪待执行的任务列表



- 合作性的循环战略
  - 调度器的周期p等于各任务周期的*最大公除数*
  - 在每个周期p，它通过调用其方法tick来推进每个任务的执行

。

- 在内部，它使用一个定时器来实现定期行为。

一个简单的调度器

```
#ifndef SCHEDULER
#define SCHEDULER

#include
"Timer.h"#include
"Task.h"

#define MAX_TASKS 10

class Scheduler {

  int basePeriod;
  int nTasks;
  任务* taskList[MAX_TASKS];
  Timer定时器。

公众。

  void init(int basePeriod);
  virtual bool addTask(Task* task);
  virtual void schedule();

#endif
};
```

Scheduler.h

```
#include "Scheduler.h"

void Scheduler::init(int basePeriod){
  this->basePeriod = basePeriod;
  timer.setupPeriod (basePeriod) ;
  nTasks = 0;
}

bool Scheduler::addTask(Task* task){
  如果 (nTasks < MAX_TASKS-1) {
    taskList[nTasks] = task;
    nTasks++;
    返回true。
  } else { return
    false;
  }
}

void Scheduler::schedule(){
  timer.waitForNextTick();
  for (int i = 0; i < nTasks; i++){
    如果 (taskList[i]->updateAndCheckTime(basePeriod)) {
                                                    taskList[i]
                                                     ->tick()。
                                                  }
                                                }
```

怡亚通-LT-UNBO

```
}
```

Scheduler.cpp

重新审视班级任务

```
#ifndef TASK
#define TASK

class Task {
  int myPeriod;
  int timeElapsed;

公众。
  虚无的init(int period){
    myPeriod = period;
    timeElapsed = 0;
  }

  虚无的void tick() = 0;

  bool updateAndCheckTime(int basePeriod){
    timeElapsed += basePeriod。
    if (timeElapsed >= myPeriod){
      timeElapsed = 0;
      返回true。
    } else { return
      false;
    }
  }

};
#endif
```

铅笔秀的例子

```
#include
"Scheduler.h"#include
"BlinkTask.h"#include
"ThreeLedsTask.h"

Scheduler sched;

void setup(){

  sched.init(50);

  任务* t0 = new BlinkTask(2);
  t0->init(500);
  sched.addTask(t0);

  任务* t1 = new ThreeLedsTask(3,4,5);
  t1->init(150);
  sched.addTask(t1);

}
```

```
void loop(){
  sched.schedule()
  。
```

RES

Ā Ā Ā Ā}

O

基于任务的架构u

铅笔秀的例子

```
#ifndef BLINKTASK
#define BLINKTASK

#include
"Task.h"#include "Led.h"

class BlinkTask: public Task {

  int pin;
  灯光*的LED。
  enum { ON, OFF} state;

public:

  BlinkTask(int pin);
  void init(int period);
  void tick();
};

#endif
```

BlinkTask.h

```
#include "BlinkTask.h"

BlinkTask::BlinkTask(int pin){
  this->pin = pin;
}

void BlinkTask::init(int period){
  Task::init(period)。
  led = new Led(pin);
  state = OFF;
}

void BlinkTask::tick(){
  switch (state){
    案关。
      led->switchOn();
      state = ON;
      break;
    case ON:
      led->switchOff();
      state = OFF;
      break;
  }
}
```

BlinkTask.cpp

铅笔秀的例子

```
#ifndef THREELEDSTASK
#define THREELEDSTASK

#include
"Task.h"#include "Led.h"

class ThreeLedsTask: public Task {

  int pin[3];
  Light* led[3];
  int state;

公众。

  ThreeLedsTask(int pin0, int pin1,
                int pin2);
  void init(int period);
  void tick();
};

#endif
```

ThreeLedsTask.h

```
#include "ThreeLedsTask.h"

ThreeLedsTask::ThreeLedsTask(int pin0, int pin1,
                             int pin2){
  this->pin[0] = pin0;
  this->pin[1] = pin1;
  this->pin[2]  =  pin2
  。
}

void ThreeLedsTask::init(int period){
  Task::init(period);
  for (int i = 0; i < 3; i++){
    led[i] = new Led(pin[i]);
  }
  状态=0。
}

void ThreeLedsTask::tick(){
  led[state]->switchOff();
  state = (state + 1) % 3;
  led[state]->switchOn() 。
}
```

ThreeLedsTask.cpp

备注

- 在任务设计中要采用隐性的强有力的纪律
  - 勾选方法的执行时间应该总是小于调度器的周期。
- 在调度器中使用定时器和中断的另一种方式是直接在中断/定时器处理程序中实现调度步骤（schedule, tick）。
  - **中断驱动的调度器**
  - 主要缺点
    - 任务执行将在中断处理程序内进行=>约束和限制

任务依赖性

- 任务可以有依赖性，需要各种形式的相互作用
  - *时间上的依赖性*
    - 例如：一个任务T3只能在任务T1和T2之后执行。
  - *生产者/消费者*
    - 例如，一个任务T1需要一个任务T2所产生的信息。
  - *面向数据*
    - 例如，任务T1和T2需要共享一些数据
- 在我们的案例中，我们通过以下方式表示和管理这些依赖关系

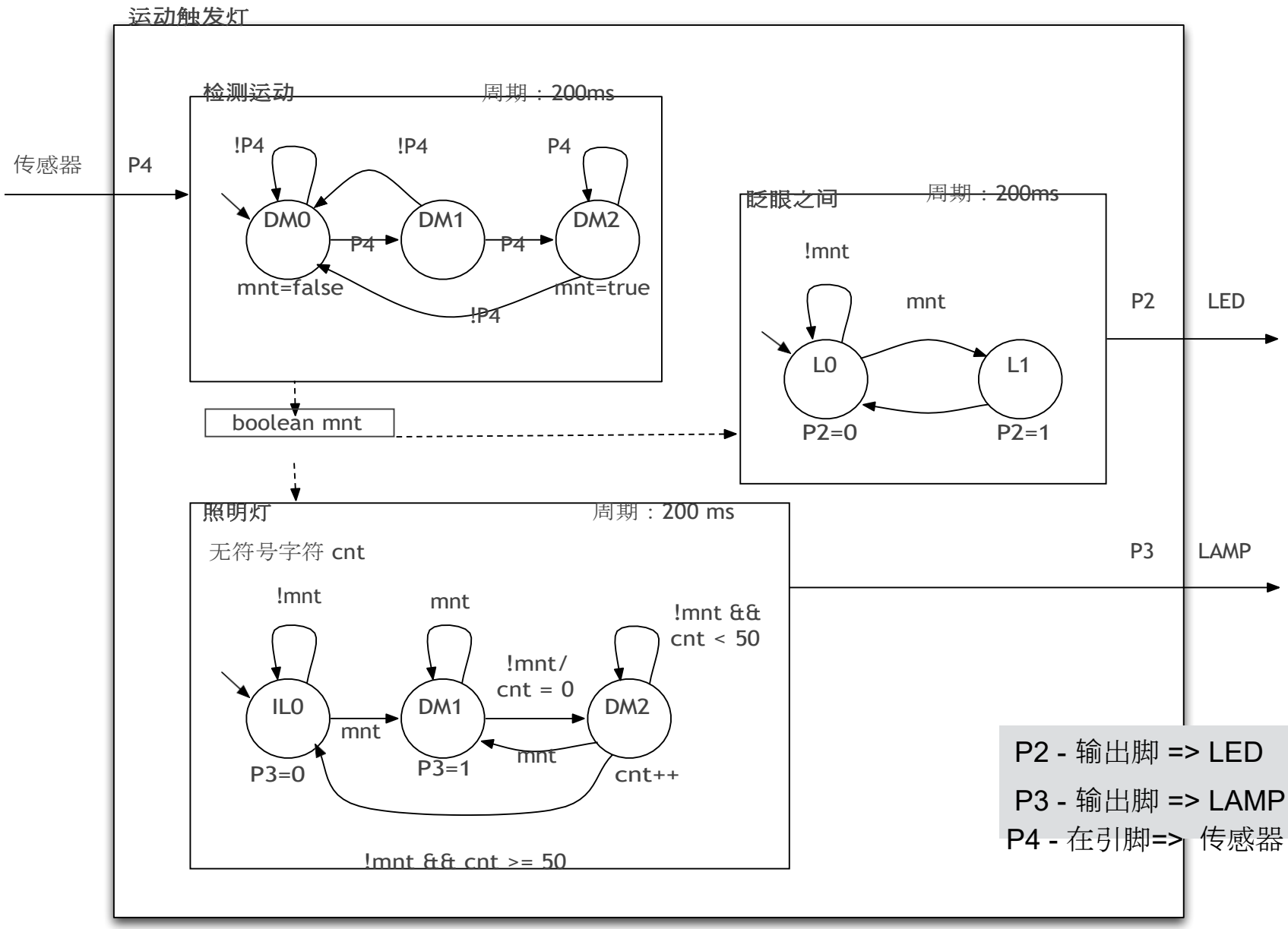
  共享变量

例子。运动触发的灯具系统

- 让我们考虑一个嵌入式系统，在这个系统中，一旦检测到某些运动，在某些环境中，一个LED就会被打开。
  - 它配备了一个运动传感器，连接在输入引脚P4上。
  - 当P4连续两次被采样为高电平，考虑到200ms的周期，就可以检测到运动。
  - 当检测到运动时，连接到输出引脚P4的LED应该被打开，并在最后的检测时间后保持10秒钟的光亮。
  - 系统包括一个连接到P2的LED，当检测到运动时（并持续检测），它应该以200ms的周期闪烁。
- 设计有三个任务
  - 检测运动
  - 幻彩灯（IlluminateLamp

– 眨眼之间

例子。运动触发的灯具系统



P2 - 输出脚 => LED

P3 - 输出脚 => LAMP

P4 - 在引脚=> 传感器

共享变量、原子行动和竞赛条件

- 一般来说，在并发任务之间共享变量可能导致竞赛条件，在并发读/写的情况下
- 在我们的案例中（合作），不可能有比赛
  - 调度器依次执行每个任务勾。
  - 从整个系统的角度来看，执行是原子性的。
- 然而，一个任务的多个时间点的执行可以与其他任务的执行交错进行。
  - 在某些情况下，这可能导致高级别的竞赛

利用睡眠模式

- 如果一个调度器的周期足够大，我们可以利用定时器驱动的行为和睡眠模式来节省电力
  - 在每个周期，调度器在执行完任务后，会进入睡眠状态，直到下一个定时器的滴答声将其唤醒，进入下一个周期。
- 这对以电池供电的系统来说是一个非常重要的扩展。
  - 然而，为了应用，调度器的周期应该大到足以使微控制器/MCU板进入/退出睡眠模式的延迟无关。

cpu使用和调度

- 在同步FSM中，我们假设动作的持续时间为零（即瞬时）。
  - 或：我们从他们的实际期限中抽象出来
- 实际上，在真实的系统中，行动总是有一个持续时间，我们需要仔细检查所选择的周期是否与该持续时间相符，以避免出现问题。

溢出异常

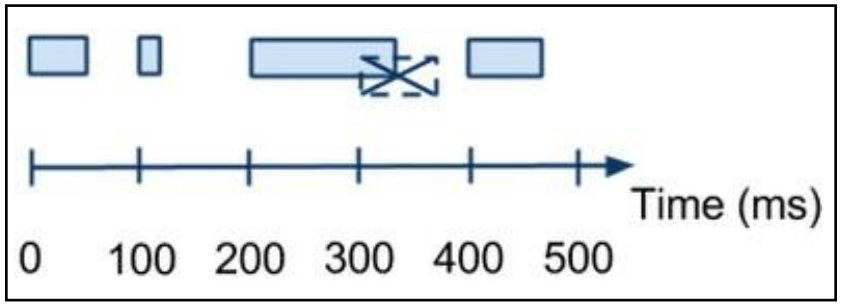
重要的是

- **超限异常--**当动作的执行时间超过周期时--如果调度器是基于定时器的使用中断，则称为*定时器超限*。

  (中断驱动的调度器)

  - 在这种情况下，一个新的中断会在中断处理程序结束前产生。

- 超限的例子



同步机，周期=100ms

根据状态的不同，执行不同的行动。在t = 200ms时，行动的持续时间超过了周期

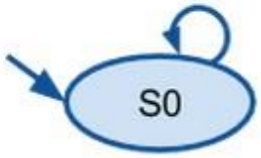
...

代码分析

- 通过对代码（汇编线）进行分析，可以发现超限异常。
  - 估计行动的总时间
  - 检查在最坏的情况下，这个持续时间是否超过期限
- 例子（[PES]，第101页）。
  - 任务CountThree只有一个状态

CountThree Period: 500 ms

unsigned char cnt;



	Estimated assembly instructions	
cnt = 0;	3	
cnt = cnt + A0;	+ 3	
cnt = cnt + A1;	+ 3	
cnt = cnt + A2;	+ 3	
cnt = cnt + A3;	+ 3	
B1 = (cnt >= 3);	+ 3 + 2	Total: 20

cpu利用率参数和最坏情况下的执行时间

- **CPU利用率参数**是指CPU（微控制器）被用来执行任务的时间百分比。

  U = （CPU用于一项任务的时间/总时间）*100%

- **最坏情况下的**执行**时间**（**WCET**）是指在最坏情况下的一段执行时间。
- 在有多个状态/过渡的情况下，我们考虑最长的指令序列

超载案例

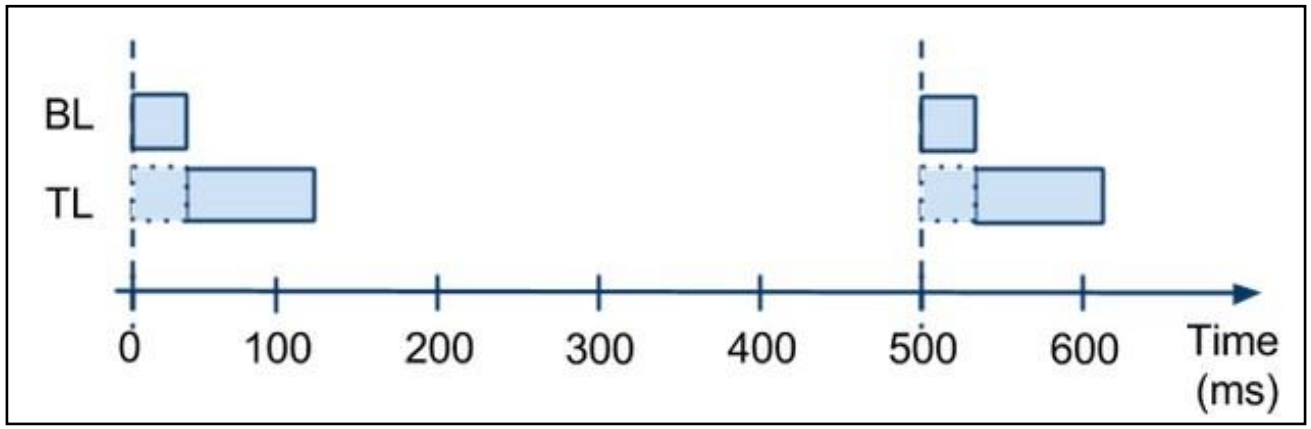
- 我们计算U，如果U>100%，则可能发生超限异常
- 在这种情况下，为了解决/避免问题，我们可以。
  - 增加FSM的周期
  - 优化指令的顺序，以减少WCET
  - 将长序列分解成小的行动序列
  - 使用更快的MCU
  - 从系统中删除功能/行为

多任务情况下的cpu利用率和wcet

- 在有多个任务的系统中，对CPU利用系数的分析甚至更为频繁
- 如果任务具有相同的周期，那么WCET的计算方法是将各个任务的WCET相加。

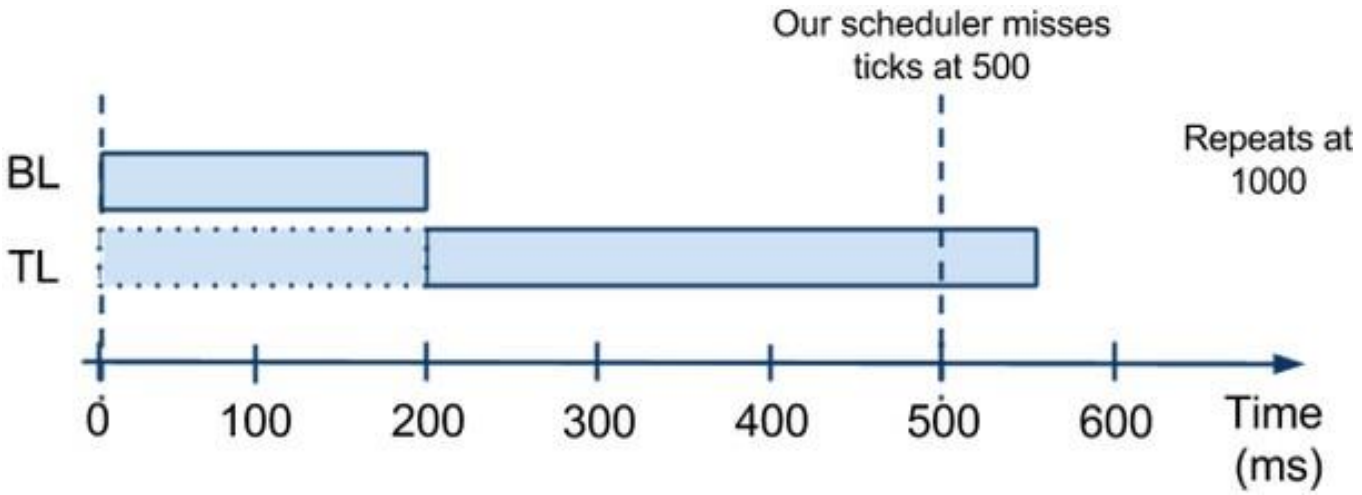
铅笔秀的例子

- LedShow例子
  - BlinkLed（BL）和ThreeLeds（TL），每个都有500毫秒的周期。
  - 分析实例
    - 假设MCU将执行100条指令/秒，即0.01秒/条。
    - BL = 3条指令的WCET => 3*0.01 = 0.03秒
    - 每个TL的WCET = 9条指令 => 9*0.01 = 0.09秒
    - U=(0.03+0.09)/0.5(0.5为周期)=24 %。



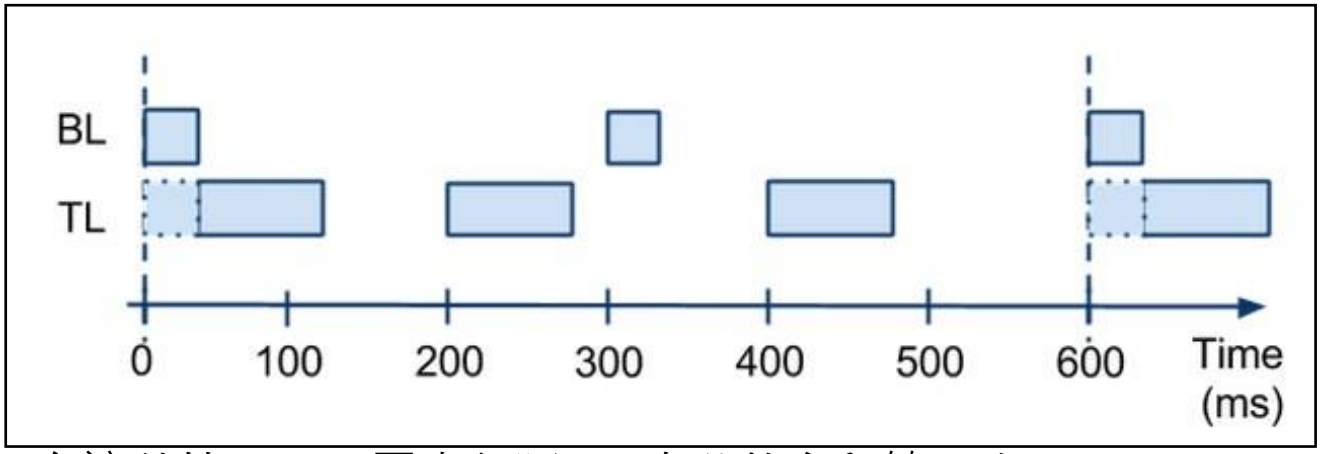
领先的显示与超限的显示

- 假设BL的WCET为200ms，TL的WCET为350
- 那么。U = (0.200+0.35)/0.5 = 1.1 => 超限



当时期不同

- 在一般情况下，任务可以有*不同的时期*
- WCET可以通过考虑**超周期来**计算。
  - 的最小公倍数的时期。
- 例子。LedShow - BL周期为300ms，TL周期为200ms



- 在这种情况下，图案每隔600毫秒就会翻转一次。
  - 在600毫秒内，BL执行了600/300=2次，而TL执行了600/200次=3次

– 超周期的U参数是：（2*20ms+3*90ms）/600ms=55%。

思考

- T_1 ...T_n 任务在单片机M上的U计算步骤
  - 我们确定MCU一条指令的持续时间R
  - 我们分析每个T的代码$_i$ ，计算其WCET
    - 我们首先计算出一个tick中的最大指令数，然后将其乘以R
  - 我们确定超周期H，作为所有任务（T1.H, T2.H,...）的最小共同乘数。
  - 然后，U被计算为

    U = ((H/T1.period)*T1.WCET + (H/T2.period)*T2.WCET+...)/H*100

    - U > 100 => 将会有超限。
    - U < 100 =>

      单个任务：不能出现超限现象 多个任务

      ：仍然可以出现超限现象

JITTER

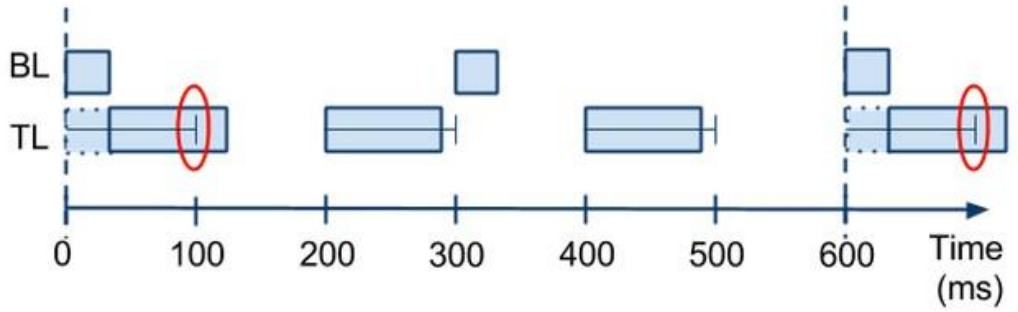
- **抖动**是指任务准备执行的时间和任务有效执行的时间之间发生的延迟。
- 不同的调度策略可能导致不同的抖动 - LedShow的例子
  - 如果BL有优先权 => BL的抖动为0，而TL的抖动为30ms。
  - 如果TL有优先权 => BL的抖动为90ms，而TL的抖动为0
- 一般来说，优先考虑最短的任务会使平均抖动最小化。
  - 例子中：优先于BL

截止日期

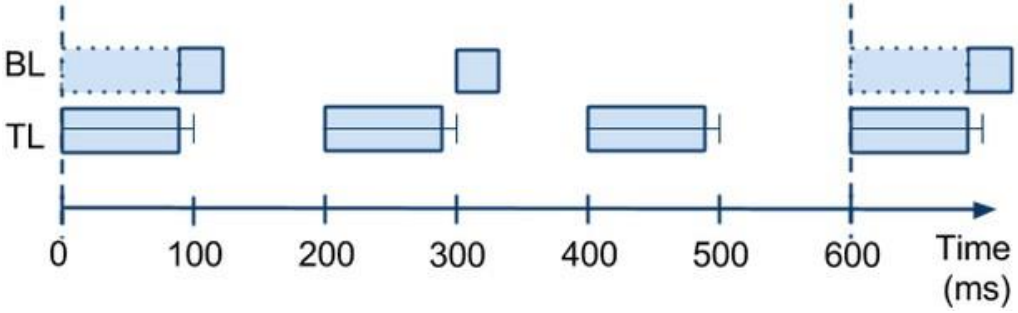
- **最后期限**（*scadenza*）被定义为一个任务在准备执行后*必须*执行的时间间隔（在每个时期）。
- 如果一个任务没有在其最后期限内执行，我们有一个***错过截止日期的例外情况***，可能会导致系统故障 - 如果没有指定截止日期，那么它就是它的期限，默认为
- 调度策略影响到紧张情绪，然后也影响到错过最后期限的可能性。

例子

- LedShow例子
  - 假设BL的周期=300ms，WCET为30ms，最后期限为300ms
  - TL的周期=200ms，WCET 90ms，最后期限100ms。
- 如果我们优先考虑BL，我们就会错过TL的最后期限。



- 如果我们优先考虑TL。



- 在这种情况下，我们有一个更大的抖动，但我们没有错过最后期限。

静态和动态优先级调度

- **优先级**涉及任务执行的顺序
  - 如果有多个任务已经准备就绪，那么调度器总是选择优先级较高的任务
  - 在分配任务的优先级方面，我们可以有静态和动态两种方法
    - 静态 => 在设计时分配优先级，在运行时不改变
    - 动态 => 根据不同的策略，动态地分配优先权
- 实时操作系统（RTOS）的关键方面
  - 下一个模块

定期和零星的任务

- 到目前为止，所考虑的任务是**定期**
  - 它们由调度器定期执行，给定一个周期p，这个周期与同步FSM的周期相对应
  - 典型的周期性任务是静态的，也就是说：在系统启动时启动，并且永远不会被移除。
- 除了周期性任务，我们还可以有**零星的或不定期的任务**
  - 即在任意时间执行的任务

管理非周期性的任务

- 管理非周期性任务的调度器的基本能力
  - 动态插入和删除任务
  - 动态分配优先权
- 简单的例子（在实验室中）
  - 带有 "空闲 "状态的周期性任务的非周期性任务模拟
    - 最简单的方法，不是超级有效
  - 具有活动/非活动元状态的任务
    - 只有在活动的情况下，才由调度员选择
    - 也可由其他任务激活/取消激活

书目

- [PES] "嵌入式系统编程--面向时间的编程介绍"。Vahid et al.