ezos简要用户手册

V0.1.0

1. ezos开发历史

ezos最早的思想起源于TCS德国的带有任务性质的超循环方式(前后台方式), 为使用方便, 作者将增减任务的方式直接和函数名相关, 加入延时功能. 并写了核心代码 ezos\_schedule(); ezos\_add(); ezos\_delete(); ezos\_delay(); 这样一个外表酷似RTOS, 内核为超循环任务的OS就诞生了, 版本V1.0.0. 由于使用起来非常方便, 代码为纯C语言, 简单易懂, 因此命名为 ezos, 即easy OS.

后续由 TCS 胡工加入了任务参数, 静态任务, 可实例化任务的概念. 整合这些概念后, 重写了部分代码, 将ezos的版本升级至较为实用的V1.3.1.在这个阶段, 使用ezos做了功能较为复杂的项目, 并发现一些任务通讯和同步的问题, 在借鉴了抢占式RTOS的一些特性后, 决心重写ezos, 将其变为真正的协作式RTOS. 同时要求代码依旧清晰简洁, 可适用于资源较少的8位单片机应用.

基于以上想法, 将ezos的提升到了V2开始的主版本. 将代码改为linux书写规范, 进化为协作式OS, 支持254级优先级. 使用了链表, 重写所有系统函数. 还有, 非常重要的是, 增加了任务间的通讯方式, 支持信号量, 互斥量, 事件, 邮箱, 消息队列.

1. ezos 原理介绍

ezos是协作式的操作系统, 即支持优先级, 支持任务间通讯. 但任务的轮询调度不是抢占式的, 而必须由任务自己放弃CPU的控制权. 即当有一更高优先级的任务需要运行时, ezos并不会马上剥夺现有任务CPU的控制权, 必须等待现有任务自己放弃CPU的控制权, 更高优先级的任务才会被运行.

一个软件系统是否需要使用操作系统, 操作系统的好坏, 个人明确强调, 没有最好只有最合适! 这是嵌入式行业的原则之一, 如果有最好最合适的, 那就成了通用的PC行业软件开发, 而非嵌入式软件开发了.

对于软件功能非常简单的嵌入式系统, 前后台/超循环模式可能就是最好的! 资源开销小, 代码直观, 易于调试.

对于有一定复杂度的嵌入式系统, 就需要使用操作系统了, 否则软件会变得难以开发, 难以理解和阅读, 难以维护和升级……

对于一般复杂度的嵌入式软件, 使用协作式的操作系统是个不错的选择, 相对于抢占式的操作系统, 其缺点为响应时间较慢(但并非不可控制). 优点为系统代码和占用资源很小, 任务流程容易预测, 便于调试和开发.

抢占式操作系统的一个最大特点是: 任务随时随地都可能被优先级更高的任务剥夺运行权, 因此也就需要程序员充分考虑任务的原子性要求, 但这是需要培训和经验积累的. 问题的严重性更在于, 此类问题很难跟踪调试, 容易让产品是不稳定的. 因此使用协作式的操作系统, 对程序员的要求较低, 也更易于开发出稳定可靠的产品.

ezos的任务数据.

ezos的任务数据存放在 ezos\_task\_t task\_link[\_\_EZOS\_TASK\_NUM];

ezos内置一个idle任务数据, 取名idle, 优先级最低, 为255. 可以简单理解为它是任务数据的表头.

当调用ezos\_add(), ezos\_delete()时, 会自行分配或释放此数据空间.

ezos核心思想是两组单向链表: 任务链表 和 回收链表.

任务链表为单向循环链表, 链表起始节点存放着系统空闲任务.后面的节点按照任务优先级的先后顺序进行排序. 对于同优先级的任务, 任务添加或任务改变自身状态时, 会将该任务节点放到同优先级链表节点的末尾.

因此, 任务链表从idle往后, 一定是按照优先级顺序排列好的!

举例来说, 任务列表如下:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 优先级1 | 优先级3 | 优先级3 | 优先级10 | 优先级11 | idle(255) |

则ezos 会按照顺序依次查询任务列表中任务的状态. 更根据状态确定是执行任务还是跳过任务. 上表中, 如果新增一个优先级为2的任务, 则此任务会被放在优先级1和优先级3之间. 如果新增一个优先级为3的任务, 则此任务会被放在优先级10之前. 另外, 如果第一个优先级3的任务运行过后, 也会被放在优先级10之前, 以便让同优先级的其它任务运行.

回收链表为单向非循环链表, 被删除的任务节点放到到回收链表中. 而被添加的任务节点从回收链表取出并分配到任务链表中.

ezos的任务状态:

1. 任务删除状态(EZOS\_DELETE), 即终止状态. 任务不在任务链表中, 不会运行.

2. 任务阻塞状态(EZOS\_SUSPEND), 任务已在任务链表中, 但被阻塞, 不会被运行. 任务延时也属于阻塞状态.

3. 任务就绪状态(EZOS\_READY), 任务已在任务队列中, 可以运行, 但未必正在运行.

4. 任务运行状态(EZOS\_RUNNING), 任务已在任务列表中, 并正在运行.

ezos任务调度

借助于ezos任务链表的良好设计, ezos任务调度非常简单, 每次都是从头开始寻找处于就绪态的任务,运行完某个任务后就从头重新开始搜寻. 由于任务链表是按照优先级顺序排序的,这样就实现了高优先级任务的快速响应. 而同优先级任务则是轮询处理的.

void ezos\_schedule(void)

{

while(1)

{

task = idle.next;

// 寻找就绪的任务

while (task->status != EZOS\_READY) task = task->next;

run = task;

run->status = EZOS\_RUNNING;

ezos\_enable\_int(mask);

// 运行任务函数

task->pfun(&task->state, &task->para);

mask = ezos\_disable\_int();

// 自动删除任务

if (run->status == EZOS\_RUNNING) ezos\_delete\_cur();

run = &idle;

}

}

由于加入了任务的命名体系, 对于可重载的函数就可以添加多个任务了.而所有的任务操作依据就是任务的名称. 任务的名称是不可重复的.对于静态任务函数, 一般将其函数名命名为任务名称即可, 方便又好记.对于可重载函数的多个任务, 建议使用 函数名+0, 函数名+1 的方式来命名任务名称.

1. ezos 使用方法

ezos 基础函数.

为了让ezos工作起来, 相关的基础函数有3个.

ezos\_init(); // ezos初始化函数, main函数尽早调用

ezos\_scan\_1ms(); // 1ms定时函数, 1ms定时中断调用

ezos\_schedule(); // ezos调度函数, 为死循环. main函数最后调用

ezos调试函数.

主要有两个调试函数.

ezos\_trash\_num\_min\_get(); // 获取系统最小的剩余任务空间

ezos\_idle\_tick\_max\_get(); // 获取idle任务运行间隔最大时间

ezos\_trash\_num\_min\_get获取系统最小的剩余任务空间. 可用来判断设置的任务数量是否合适, 太小会让任务添加失败, 太大会造成RAM的浪费(但不会造成运算效率的浪费). 在稳定运行系统一定的时间后, 根据观察结构, 调整 \_\_EZOS\_TASK\_NUM 值即可, 建议根据结果, 再预留2-5个任务较为合理

ezos\_idle\_tick\_max\_get获取idle任务运行间隔最大时间. 可用来判断CPU是否任务过重, 以及任务切换最慢响应时间(最低优先级任务的最差情况).

而最高优先级任务的最差情况. 根据协作式的特点分析可知:

最高优先级任务的响应时间 = 某个独占CPU时间最长的ezos任务

因此, 在ezos任务中, 不要使用独占CPU的长延时!!! 如果任务过重, 可手动拆分成多个小任务, 完成一次小任务就主动放弃一次CPU的控制权.

常用的任务函数.

ezos\_t ezos\_add( void \*name, // 任务名称, 该任务的唯一标识

void (\*pfun)(ezos\_s\_t\*, ezos\_p\_t\*), ezos\_p\_t para, // 任务函数

int32\_t delay\_time, // 任务延时时间

uint8\_t priority); // 任务优先级

quick\_add(); // 快速添加任务, 用的较多, 输入参数只需pfun和priority.

force\_add(); // 强制添加任务, 会先删除该任务, 再重新添加

ezos\_delay(int32\_t time);

常见的任务编程方式.

1. 单次任务, 见范例 app\_lcd之task\_lcd
2. 循环任务, 见范例 app\_rtc之task\_rtc\_1s
3. 状态机任务, 见范例 app\_race\_led 之task\_race\_led
4. 复杂任务, 见范例 app\_bell之task\_bell\_1s
5. 可重入任务, 见范例 app\_sem之task\_led
6. 任务的拆分, 见范例 ITT100 dv\_pic之SavePIPtoSPI
7. ezos之任务通讯

ezos的任务通讯相关函数统一放在ezos\_ipc.c文件下. ezos会将所有的添加的ipc串成一个链表. 其数据结构如下:

struct \_\_ezos\_ipc

{

void \*next; // 下个IPC指针

void \*name; // 当前IPC指针

ezos\_t type; // IPC类型

};

typedef struct \_\_ezos\_ipc ezos\_ipc\_t;

此数据结构相当于ipc的父类, 基于此父类, 下面会衍生出各个IPC子类,

如信号量:

typedef struct

{

ezos\_ipc\_t ipc; // IPC父类

uint8\_t value; // 信号量值

} ezos\_sem\_t;

所有IPC的添加与删除都是对此链表的操作, 使用函数为:

ezos\_t ezos\_ipc\_add(void \*name, ezos\_t type)

ezos\_t ezos\_ipc\_delete(void \*name)

IPC的使用

IPC的释放/发送较简单, IPC的获取/接收使用稍有复杂, 主要是需要判断IPC的返回值, 并进行相关操作.此处已信号量(sem)为例.

原函数见:

ezos\_t ezos\_sem\_release(ezos\_sem\_t \*sem)

ezos\_t ezos\_sem\_take(ezos\_sem\_t \*sem, int32\_t timeout)

对于 ezos\_sem\_take, 返回值种类有4种! 依次为:

1. EZOS\_OK, 获取信号量成功

2. EZOS\_WAIT, 需要等待信号量, 返回此值后, 原函数需要return

3. EZOS\_TIMEOUT, 信号量等待超时

4. EZOS\_ERROR, 指定信号量有错误, 说明代码有误

使用信号量时, 基本格式为:

EZOS\_TASK(task\_sem)

{

ezos\_t val;

val = ezos\_sem\_take(lock, 10);

if (val == EZOS\_WAIT) return; // 直接返回等待

if (val == EZOS\_OK) {do\_something();} // 成功获取信号量

else if (val == EZOS\_TIMEOUT) {do\_something();} // 信号量超时处理

else if (val == EZOS\_ERROR) {exception();} // 异常处理

}

IPC编程方式.

1. 信号量(sem), 见范例 app\_sem
2. 互斥量(mutex) 及 事件(event) 与 信号量类型, 略
3. 邮箱(mailbox), 见范例 app\_mail
4. 消息队列(message), 见范例 app\_message