# 线程的实现

RTT设备驱动的意义：

解决驱动碎片化问题，将驱动与应用解耦。简单的例子来说，今年我再ST的芯片上写了10w+的应用代码，发现这个芯片的性能不能满足我后面项目的需求了，假如有设备驱动的支持，可以将10w+的代码无缝切换的其他公司的芯片上，而改动是重写一遍驱动

多线程系统事件的响应也是在中断中完成了，但是事件的处理是在线程中完成的。线程跟中断一样，也具有优先级。

线程： 我们根据功能把整个系统分割成一个个独立的且无法返回的函数。 这个函数我们称之为线程。

定义线程栈：

多线程中，每个线程是独立的，互不干扰的， 所以要为每个线程分配单独的线程栈。 这个栈空间可以是预先定义的全局数组，也可以是动态分配的内存。

设置字节对齐可以提高内存的访问效率。

定义线程控制块：

线程是由系统调用的，系统为了顺利的调度线程， 为每个线程定义了额外的线程控制块。

相当于线程的身份证， 包含线程的堆栈指针， 线程名称，线程参数等等。 对线程的操作可以通过线程控制块来完成。

初始声明：

Struct rt\_thread

{

void \*sp； /\*线程堆栈指针\*/

void \*entry； /\* 入口地址\*/

void \*para; /\*形参\*/

void \*stack\_addr; /\* 栈开始地址\*/

void \*stack\_size; /\* 线程栈大小\*/

rt\_list\_t tlsit; /\* 线程链表节点\*/

};

Typedef stuuct et\_thread \*rt\_thread\_t;

线程创建函数：

rt\_thread\_init(struct rt\_thread \*thread,void(\* entry)(void \*para),void\*para,void \*stack\_start,rt\_uint32\_t statck\_size)

函数的实现：

1. 初始化线程链表节点： rt\_list\_init(&(thread->tlist))

双向链表rt\_list\_t 的定义：

struct rt\_list\_node

{

struct rt\_list\_node \*next; /\*指向后一个节点\*/

struct rt\_list\_node \*prev; /\* 指向前一个节点\*/

}

链表的初始化： 双向的环形链表

链表的初始化就是把节点里的next和prev这两点节点指针指向节点本身。

双向无头循环链表参考代码：

<https://www.cnblogs.com/chengzi123/p/4340471.html>

#include <stdio.h>  
#include <stdlib.h>  
#define OVERFLOW -2  
#define OK 1  
#define ERROR 0  
  
//此双向循环链表无头结点  
typedef int ElemType;  
typedef struct DulNode {  
    ElemType data;  
    struct DulNode \*prior;  
    struct DulNode \*next;  
}DulNode,\*DulLinkList;  
  
DulLinkList InitList\_Dul(DulLinkList L);  
DulLinkList CreateElemP\_Dul(DulLinkList L);  
void PrintList\_Dul(DulLinkList L);  
int GetElemP1\_Dul(DulLinkList L,int i,ElemType \*e);  
DulLinkList ListInsert\_Dul(DulLinkList L,int i,ElemType e);  
int ListLength\_Dul(DulLinkList L);  
DulLinkList GetElemP2\_Dul(DulLinkList L,int i);  
DulLinkList ListInsert\_Dul(DulLinkList L,int i,ElemType e);  
DulLinkList ListDelete\_Dul(DulLinkList L,int i,ElemType \*e);  
int GetLocate\_Dul(DulLinkList L,ElemType e);  
int main()  
{  
    DulLinkList dul;  
    int len,loc;  
    ElemType e;  
    dul = InitList\_Dul(dul);  
    dul = CreateElemP\_Dul(dul);  
    len = ListLength\_Dul(dul);  
    printf("该双向链表的长度为：%d\n",len);  
    PrintList\_Dul(dul);  
    GetElemP1\_Dul(dul,3,&e);//3为元素所在的位序  
    printf("第i个位置的元素为：%d\n",e);  
    printf("向链表中插入一个元素后：");  
    dul = ListInsert\_Dul(dul,1,0);//1为插入元素的位置  
    PrintList\_Dul(dul);  
    printf("从链表删除一个元素后：");  
    dul = ListDelete\_Dul(dul,2,&e);//2为删除元素的位序  
    PrintList\_Dul(dul);  
    printf("删除的元素是：%d\n",e);  
    loc = GetLocate\_Dul(dul,0);//0为元素的值  
    printf("该双向循环链表元素i的位序为:%d\n",loc);  
    return 0;  
}  
  
//初始化一个双向循环链表  
DulLinkList InitList\_Dul(DulLinkList L) {  
    L = (DulLinkList)malloc(sizeof(DulNode));  
    if(L != NULL) {  
        L->next = L->prior = L;  
    } else {  
        exit(OVERFLOW);  
    }  
    return L;  
  
}

//创建一个双向循环链表  
DulLinkList CreateElemP\_Dul(DulLinkList L) {  
    int n,i;  
    DulLinkList p,s;  
    s = L;  
    printf("请输入双向循环链表的元素个数：");  
    scanf("%d",&n);  
    printf("请输入双向循环链表中元素的值：");  
    for(i=0; i<n; i++) {  
        p = (DulLinkList)malloc(sizeof(DulNode));  
        scanf("%d",&p->data);  
        p->prior = s->prior;  
        s->prior->next = p;  
        p->next = s;  
        s->prior = p;  
    }  
    return L;  
}  
  
//输出该双向循环链表  
void PrintList\_Dul(DulLinkList L) {  
    int i;  
    DulLinkList p;  
    p = L;  
    printf("该循环链表中的元素为：");  
    while(p->next != L) {  
        p = p->next;  
        printf("%d ",p->data);  
    }  
    printf("\n");  
}  
  
//返回双向循环链表的长度  
int ListLength\_Dul(DulLinkList L) {  
    DulLinkList p;  
    int i;  
    p = L->next;  
    i = 0;  
    while((p!=L) && (p!=NULL)) {  
        ++i;  
        p = p->next;  
    }  
    return i;  
}  
  
//取得双向循环链表上第i个位置上的元素的值  
int GetElemP1\_Dul(DulLinkList L,int i,ElemType \*e) {  
    DulLinkList p;  
    int j;  
    p = L;  
    j = 0;  
    while(p->next && j<i) {  
        ++j;  
        p = p->next;  
    }  
    if(!p || j>i) return ERROR;  
    \*e = p->data;  
    return OK;  
}  
  
//返回双向循环链表上第i个位置上的指针  
DulLinkList GetElemP2\_Dul(DulLinkList L,int i) {  
    DulLinkList p;  
    int j;  
    p = L;  
    j = 0;  
    while(p->next && j<i) {  
        ++j;  
        p = p->next;  
    }  
    if(!p || j>i) return ERROR;  
    return p;  
}  
  
//向该双向循环链表的第i个位置之前插入元素e  
DulLinkList ListInsert\_Dul(DulLinkList L,int i,ElemType e) {  
    DulLinkList p,s;  
    int m;  
    if(!(p=GetElemP2\_Dul(L,i))) {  
        return ERROR;  
    }  
  
    if(!(s = (DulLinkList)malloc(sizeof(DulNode)))) {  
        return ERROR;  
    }  
    s->data = e;  
    s->prior = p->prior;  
    p->prior->next = s;  
    s->next = p;  
    p->prior = s;  
    return L;  
}  
  
//删除双向循环链表上第i个位置的元素，并返回该元素的值  
DulLinkList ListDelete\_Dul(DulLinkList L,int i,ElemType \*e) {  
    DulLinkList p;  
    if(!(p=GetElemP2\_Dul(L,i))) {  
        return ERROR;  
    }  
    \*e = p->data;  
    p->prior->next = p->next;  
    p->next->prior = p->prior;  
    return L;  
}  
  
//返回双向循环链表上元素e的位序  
int GetLocate\_Dul(DulLinkList L,ElemType e) {  
    DulLinkList p;  
    int j;  
    p = L->next;  
    j = 0;  
    while((p != L) && (p->data!=e)) {  
        j++;  
    }  
    return j;  
}

/\* 初始化线程栈\*/

Rt\_hw\_stack\_init()函数

传入栈顶指针，堆栈8字节对齐（方便浮点运算），预先设置线程第一次运行时，需要加载到CPU寄存器中的参数。

创建一个线程需要的内容： 线程控制块，线程的入口地址， 线程的形参， 线程堆栈起始地址，堆栈大小。

# 就绪列表的实现

线程创建完毕会添加到就绪列表， 表示线程已经就绪，可以随时进行调度，就绪列表在文件scheduler.c中定义。

就绪列表的实质是一个rt\_list\_t 类型的数组，数组的大小由决定线程最大优先级的宏

RT\_THREAD\_PRIORITY\_MAX 决定。 数组的下标对应了线程的优先级。

将线程插入到就绪列表：

线程控制块中有一个tlist成员。 数据类型是rt\_list\_t, 插入就绪列表就是把tlist成员加入就绪列表。

# 调度器的实现

操作系统的核心，主要实现线程的切换。 即从就绪列表中找到优先级最高的线程， 然后去执行该线程。

调度器使用之前需要初始化。

初始化： 初始化线程的就绪列表（初始化为0 ）。初始化当前的控制块指针为NULL。

启动调度器： rt\_systerm\_scheduler() 来实现。

调度器在启动时从就绪列表中取出优先级最高的线程的控制块。 （具体实现由就绪列表推算出控制块的地址）。然后切换到该线程。

线程切换：

Rt\_hw\_context\_switch\_to() 函数

汇编函数实现的：

EARA，表示新的数据段或者代码段。

.test 段名字

PORC 用来定义子程序，ENDP成对使用。

PenDSV\_handle（）函数。

真正实现线程上下文切换的地方：

过程： 关中断（防止切换过程被打断）->获取中断标志位，如果不为0 ，继续执行，为0 则退出异常。线程切换的实质是更改PC指针和PSP指针。 加载rt\_interrupt\_to\_thread 的值到r1。更新r4~r11,更新PSP的值。 如果线程不是第一次切换，需要先保存上文内容。

切换时，上一个线程的的运行环境，xPSR, PC ,R14,R12,R3,R2,R1,R0等CPU 的值会自动保存到线程栈中。 R4~r11需要手动保存。

系统的调度：

记录前一个线程控制块，更改到当前程序控制块。

Rt\_hw\_contex\_switch()函数来实现。

当一个汇编函数在C文件中调用时，如果有两个形参，在会将这两个形参传入r0，和r1。

如果彻底搞懂，需要查看M3/M4/M7的内核指南。

问题点：

1. PSP
2. SP

# 临界段保护

临界段： 在执行过程中不能被中断的代码段。

在操作系统中，临界段保护最长出现的是对全局变量的操作。

在RTT\_Thread 中，系统调度会产生PenSV中断。 在PenSV Handler 中实现上下文的切换。

所以RT\_thread对临界段的保护很干脆，直接关闭所有的中断。

临界段代码的应用：

进入临界段代码之前，先把中断关闭，退出后，开启中断。 需要保存PRIMASK的值，防止临界段嵌套出现问题。

# 对象容器的实现

在RT\_Tread中，所有数据结构都称之为对象。

在rtdef.h中有定义。 包含线程，信号量，互斥量，事件，邮箱，消息队列，内存堆，内存池，设备，定时器，模块，静态对象等。

内核对象的定义： 对象名字， 对象类型， 对象的状态，对象的列表节点， 对象的数据类型

对象的列表节点是为了方便讲自己挂到对象容器中。

在RT\_Thread中，每个对象都有一个结构体，叫做该对象的控制块。 比如线程控制块。

容器： 在rtt中，每当用户创建一个对象，都会把对象放到容器中。这样做的目的是为了方便管理。

容器是一个数组， 是一个全局变量。数据类型为 struct rt\_object\_information.

结构体内容包含：对象类型，对象列表节点头，对象的大小。 在rtdef.h中定义。

容器在定义时，大小是被固定的，由RT\_Object\_Info\_Unknown这个枚举值决定。

对象的初始化： 将成员初始化，将该对象插入到对象容器中。

对象初始化函数在线程初始化函数里面被调用。

# 空闲线程和延时阻塞

线程如果需要延时的时候，会放弃CPU的使用权。 叫做延时阻塞。

在阻塞延时时。如果没有其他需要运行的线程，会创建一个空闲的线程。

空闲线程是系统初始化的时候自动创建的优先级最低的线程。

空闲线程的栈是一个定义好的数组，大小由IDLE\_THREAD\_STACK\_SIZE 的宏控制，默认512字。

RTT的空闲线程主要做对内存的清理工作。

在系统初始阶段会对空闲线程进行初始化，然后插入到就绪列表的末尾。

延时阻塞的逻辑：

1. 获取当前的时间
2. 设定延时的时间
3. 进行系统的调度

Remaining\_tick 是线程控制块的一个成员，目的是为了记录需要延时的时间。

SysTick\_handle中断服务函数：

主要更新时基

在系统调度函数rt\_schedule() 中， 会判断每个线程的线程控制块中的Remaining 的时间是否为0 ，如果为0，就将对应的线程就绪

操作系统最小的时间单位SysTick的中断周期。

在进行线程调度之前，需要先将中断关闭。

# 多优先级的实现

就绪列表由两个在scheduler.c 文件中定义的全局变量来实现。

一个是线程就绪优先级数组，另一个是线程优先级表。

线程优先级表： 线程启动的时候要根据优先级来决定插入到优先级表的位置。

为了快速找到线程在线程优先级表的插入和移除位置， RT-Thread 专门设计了一个线程就绪优先组。 从代码上看，线程就绪优先组就是一个32位的整形数。每一个位代表一个优先级。一个就绪优先组最多包含32个优先级。 线程就绪优先组的每一个bit对应一个优先级，位0对应优先级0 ，位10 对应优先级10。 根据优先级的位数快速索引线程优先级表。

线程优先级表: 一个rt\_list\_t 数据类型的数组。 每个元素是一个双向环形链表。 表示相同优先级的几个任务。

调度器插入线程，也就是，把线程插入到就绪列表。 同时改变线程就绪优先组中对应的bit。

线程就绪优先组为0，代表还没有一个程序就绪。

# 定时器的实现

/\* 如果人生没有追求，那么和咸鱼有什么区别呢\*/

在RT\_thread 中，每个线程都内置一个定时器，当线程需要延时的时候，将线程挂起，然后内置的定时器启动。 并且将定时器插入到一个全局的系统定时器rt\_timer\_list 中。Rt\_list\_t 是一条双向链表。 每个节点代表了正在延时的定时器。 节点按照延时的大小升序排列。

这样在每次时基中断到来的时候，就稻苗系统定时器的第一个定时器就OK了。

定时器统一由一个定时器结构体来管理。 成员包括父节点，本身节点，超时函数， 延时时间，定时器实际超时是的系统节拍数。为了精确和校正。

定时器也属于内核对象。

在线程控制块中包含定时器成员。

定时器删除就是从rt\_timer\_list 中脱离即可。

定时器删除： 将定时器从系统定时器列表中删除，然后改变定时器的状态为非active即可。

定时器控制函数： rt\_timer\_control() 在timer.c中定义， 用来设置定时器的状态和初始时间的值。

定时器启动函数： rt\_timer\_start 在timer.c 中定义，核心算法将定时器按照延时时间做升序加入系统定时器rt\_timer.c 中。

定时器扫描函数： rt\_timer\_check在timer.c中定义。 用于扫描系统定时器列表。 查询定时器的延时是否到期。

# 时间片的实现

同一优先级支持时间片轮转。

# RT\_Thread 应用

裁剪RT\_Thread 的功能，可以通过修改 RT\_Thread 的内核配置文件来裁剪RT\_Thread 的功能。

在RT\_Thread看来，除了内核，其他的第三方加进去的软件都是组件。 比如GUI，比如fatfs，lwip，finsh等等。 Finsh 组件是RT\_Thread 组件中最具特色的，通过串口打印各种调试的信息。

Libcpu: 软硬件的关联。 接口文件。 通常由RT\_Thread 和 mcu原厂的人负责。

Src: 文件 内核文件。 （source （存放src文件）和 ports （存放board.c文件））。

Board.c ：rt\_hw\_board\_init() 的函数，作用用来初始化开发板硬件。 需要用户编写。

SysTick() 函数：可以用固件库函数来替代。

硬件BSP初始化。

内存初始化。

应用流程：

1. 修改board.c 函数： 修改内容，屏蔽SysTick 相关的寄存器和初始化函数。 由固件库里的函数进行替代。 如果使用CubeMX生成。我们只能使用HAL\_SYSTICK\_Config重新初始化。
2. 添加core\_delay.c 文件