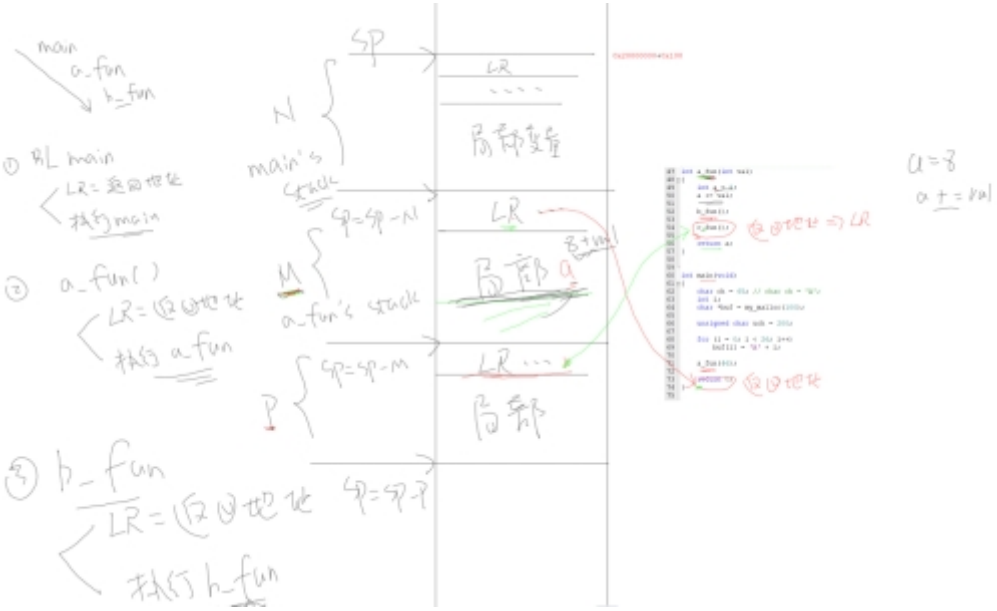
Freertos学习

栈

栈里面会放返回地址和局部变量，来一个新函数就会往下多占一段空间然后继续放新函数的返回地址和局部变量；



rtos中每个任务都有自己的栈

- 位操作

  第x位置1；

  num = num | 1 << n;

  第n位清0：

  num = num |~(0 << n)

  测试第n位：

  if(num & 1 << n)

  从第n位开始，取m位。num=（num|m个1<<n)>>n

堆

堆前面会有长度信息，要注意栈大小够不够

任务的创建

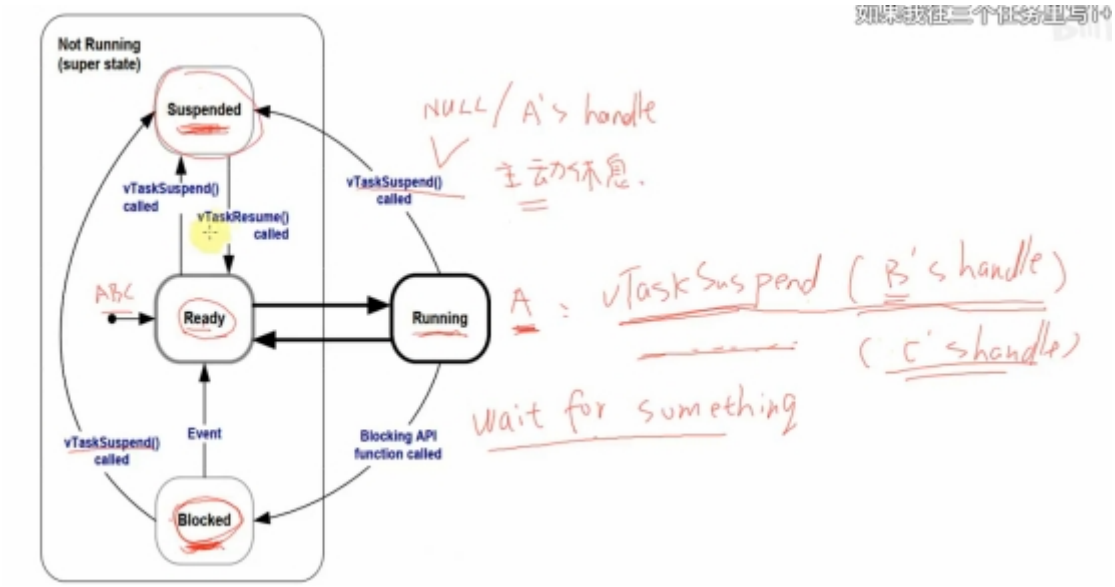
- 动态 创建任务

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode, // 函数指针，任务函数
 const char * const pcName, // 任务的名字
 const configSTACK_DEPTH_TYPE usStackDepth, //栈大小,单位为 word,10 表示40 字节
 void * const pvParameters, // 调用任务函数时传入的参数
 UBaseType_t uxPriority, // 优先级
 TaskHandle_t * const pxCreatedTask ); // 任务句柄，以后使用它来操作这个任务
```

- 静态创建任务（静态的话，TCB结构体要提前分配好，栈要提前分配好）

  xTaskCreateStatic()

任务状态



存在不同的状态链表

```
vTaskDelayUntil( pxPreviousWakeTime, xTimeIncrement ); //从第一个参数的时间，延迟固定时间结束
```

```
void vTaskDelay( const TickType_t xTicksToDelay); // 从结束到开始的固定延时
```

任务把自己杀掉，必须有空闲任务清理内存。

启动调度器时会帮你创建空闲任务

调度策略

调度策略：确定哪个就绪态的任务可以切换为运行状态。

- 抢占
- 时间片轮转
- 空闲任务是否礼让

队列

像一个传送带，先传送的先到达；

创建，写，读 以传递数值，消耗空间

创建：

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

锁的实现

> 调用读队列函数（第三参数设置一直等待直到有数据可读），获得锁使用权，之后队列里面无数据，其他用户无法读到数据只能阻塞
>
> 再调用写队列消息函数释放使用权。（队列里面有数据表示别人可以读这个队列）

队列可以传递 指针（malloc，free） 字符 整型结构体等，通过定义不同类型的变量放入第二个参数来控制。

信号量

是一个计数值 （相比队列可以省空间，但是不能传递数值）

生产者让计数值++，消费者让信号量--，计数值为0时，消费者可以进入阻塞以等待。

分为：计数型和二进制型

用法 creat give take

互斥量

队列的问题：优先级继承、递归锁。互斥量可以解决

- 创建时发送一个1
- 自带优先级继承
- 但是无法解决谁持有，谁释放的问题

递归锁：

解决谁持有，谁释放的问题，非持有锁的人无法释放

事件组：

生产者任务可以设置某一个事件产生了，消费者可以等待一个或者多个任务

信号量同步这些可以减少cpu资源浪费

全局变量，一直检测 浪费资源

中断管理

用于ISR的API函数 如果发送不了，直接返回不阻塞、不进行函数调度(但记录)

资源管理

临界资源管理（互斥）

任务a和任务b都会用一个资源:开启任务之后就关闭任务调度，结束任务的时候恢复

- 任务a和中断b：在任务中屏蔽中断；
- 在中断中，记录中断状态，屏蔽中断，恢复中断状态