app_kernel简史

阅读和跑程序的过程中，遇到任何问题或bug，请联系我：

微信：GuiStar_IUaena930516

致谢：

感谢生我养我的父母，感谢一直帮助我的亲姐姐，感谢老师的教导，感谢带我打比赛的学长，感谢学校的智电科创社..... 感谢人生中遇到的每一位有缘人，有默默支持我的，有与我并肩作战的，有驱使我不断成长的，有磨练我性格的.....我的大学四年，并不算太精彩，但是也心满意足了，能有缘与各位相见，是我莫大得福气！也感谢我人生中实习的第一家公司，这个项目百分之50的代码都是在实习期间写的，百分之八十的bug也是在实习期间解决的，尤其重要的是，感谢读者，愿意花废宝贵的时间来阅读此文，相信我，app_kernel一定不会让你失望！

app_kernel名字的由来：

app_kernel这个名字的初次出现是我打的某一个比赛，写的一个函数的名字叫“app_kernel”，app解释为应用，kernel解释为核心（或内核），这个名字起地很好，因为这个函数当时确实是实现比赛赛规要求的最顶层的一个函数，由它来负责调用之前写好的所有各个独立模块驱动代码，最终控制一个小车完成比赛的复杂要求，所以它扮演着整个系统的最高级别的应用业务逻辑控制，所以叫它app_kernel不过分，因为他确实是应用层面的核心。

背景：

rtos功能概述：

rtos的特性是多任务，因此使用它最关键的一点就是同步任务与任务，任务与中断的工作。为了做这些工作，rtos一般都做了以下几个内核对象以支持上述工作：

信号量，互斥量，临界区，事件等等

通过利用这些内核对象的机制，我们就可以完成大部分同步工作以正确的允许一个rtos系统了。举个例子：

同步中断与任务，一般会采用二值信号量：创建一个二值信号量，将其默认值设置为0，然后创建一个任务，这个任务试图获取这个二值信号量而陷入阻塞态；此时如果中断到来，那么释放这个二值信号量。刚刚阻塞的任务由于成功获取到二至信号量而得到运行，这是一个典型的中断与任务的同步。

以抢占式调度的rtos为例，rtos主要负责完成任务间的同步，通信等功能，并且每个任务都可以根据系统调度进入挂起，阻塞，运行，就绪等等任务状态来达到完成用户复杂功能需求的目的。

另外，有些系统还有软件定时器，钩子函数等特殊特性，甚至针对单片机的外设专门定制的rtos支持级别的外设。

传统rtos存在的问题：

但是上述工作要写的代码量有点过多了（个人认为）

如果需求再复杂一点呢？比如在中断中触发事件，然后要求过了500ms再执行某个回调函数，这样子可能要写更多代码，甚至还需要配合软件定时器或者系统滴答计数器，确实挺麻烦的。站在应用的角度，这个需求其实很简单，就是中断触发某个事件，然后过500ms之后去处理这个事件。

根据上述例子，举一个实际应用场景：比如在中断或某个任务中触发某个事件了，然后要求现在立刻让蜂鸣器响500ms，然后不响，在这500ms期间，单片机还要干其他的重要任务，不能死等着

500ms。这个需求很常见的，我看我们学校打的智能车比赛，人家就很喜欢到达一个路口就蜂鸣器响一下，但是智能车还是接着很快的跑着

上述例子，涉及到了中断与任务同步，任务与任务同步再结合上对时间的处理，写这些代码如果仅仅用裸机或rtos的内核对象的话，也可以写出来，但是将会很费劲，更重要的是，让其他人阅读理解起来也不太直观，那该怎么解决呢?

app_kernel的诞生：

为了改善freertos的调用机制，我创造了app_kernel这个模块。

解决上述问题的办法就是写一个模块，这个模块的主要目标就是：将目光先集中在如何简化rtos的应用级别的代码上。比如我刚刚举的蜂鸣器的例子，当中断或任务触发了某个事件，此时我想让系统干的事情是执行蜂鸣器响500ms然后不响，并且这500ms不能影响单片机的中断和其他的任务。这个过程有以下几个要点：第一，触发某个事件后，才让响蜂鸣器500ms；第二，蜂鸣器响的期间不能影响中断和其他的任务；第三，触发某个事件后，要立刻返回，不能等太久，因为此时在中断上下文!

所以我写的模块要有以下的特性：

1.调用函数1之后，系统会立刻在某个任务中调用它指定的某个回调函数。

2.回调函数里面有500ms延时操作。

3.函数1必须是非阻塞的，也就是说调用它的过程必须瞬间完成。

这样说可能比较抽象，我直接把代码贴出来就不抽象了：

```
void beep(void* param, uint32_t argc){
     beep_on();     //开启蜂鸣器
     osDelay(500);  //延时500ms，注意这里必须使用rtos的延时接口函数
     beep_off();    //关闭蜂鸣器
}

//定义signal_t类型的全局变量
signal_t signal;
void main_task(void){
    connect(signal, beep); //连接signal与beep函数
    while(1){
      vTaskDelay(1);
    }
```

```
14  }
15
16  //下面这个函数是中断函数
17  void interrupt1_function(void){
18      //触发中断函数之后
19      //发送信号通知内核马上准备调用signal1的回调函数beep
20      emit_FromISR(signal);
21  }
22
```

上述代码是伪代码， 事先定义好一个全局变量signal，然后在任务中调用一个函数connect连接signal与beep函数，简单理解为给两者建立某种关系，有了这种关系之后，后续在中断函数interrupt1_function中调用emit_FromISR发送signal，之后就会立即在合适的时机在任务上下文调用beep函数，并且注意，在中断函数里面要快尽快出的，这是一个嵌入式系统是否合格的基本评定，所以这就要求发送信号函数emit_FromISR要立刻发送完毕信号，这一点它能做到的。

这样的接口就很简单了，比直接使用rtos的内核对象简单很多，但是这只是一个接口描述，实际上，rtos并没有提供这样的接口，那就自己写一个这样的接口。这就是我做的一些工作，我开发了一个基于freertos 的子模块-app_kernel，它提供了上述的接口，在app_kernel模块里面我叫做“signal and slot”，即“信号与槽”，熟悉Qt的朋友听到这里可能已经激动了。没错，app_kernel提供了上述接口，异步实现了Qt里面的信号与槽的功能。所谓异步实现，即发送信号函数执行过程非阻塞（瞬间调用完毕），发送信号之后，不会立即调用槽函数，而是由app_kernel的内部线程决定什么时候在任务上下文调用槽函数，当然了，肯定是越及时越好的。这一点请放心。

在开发阶段我在网上看到了一个同步实现“信号与槽”的方式，那个作者的方法是直接在发送函数里面调用槽函数，这就导致发送函数的调用时长取决于槽函数。并且不能在中断里面调用。。。总之，他这种方式问题很大很大。嗯，确实很“同步”，直接在信号发送里面调槽函数了，那你为什么不直接调槽函数呢？为何要多嵌套一个信号发送函数呢？这样难道不会因为无意义的入栈出栈浪费效率吗？总之，这种同步实现的写法没任何意义！

以上只是app_kernel的其中一个应用场景，实际上，它除了signal这个机制之外，还有其他的一些机制，它们都旨在简化freertos的使用，当然，要看用户如何看待这种简化，可能不同的人有不同的想法，会觉得这样反而复杂了不易理解...

"信号与槽"只是app_kernel的其中一个组件，app_kernel还有time，call这两个主要组件，以及其他的一些小功能比如：提供系统时基，非阻塞延时函数，心跳灯等等。通过示例程序以了解更多信息。

app_kernel只有两个文件：app_kernel.c，app_kernel.h，代码量在1200行左右，平均每个组件300行代码。运行内存最低要求10000bytes约10k左右，比较轻量级。它构建于freertos系统之上，因此使用它必须先有一个freertos的单片机工程。（这里推荐一下stm32单片机的cubemx工具，一键配置freertos工程，非常方便快捷，纵享丝滑地体验app_kernel）如果你恰好有一个freertos工程，并且有对应的的单片机开发板引出来了任意一路串口，则参照app_kernel.h的注释即可非常轻松的把它移植到你的开发板上体验app_kernel。

当然如果没有freertos工程也不要灰心，仍然可以参照我的视频使用stm32单片机的cubemx快速搭建一个freertos工程并移植好app_kernel。

移植过程遇到任何问题可联系我。

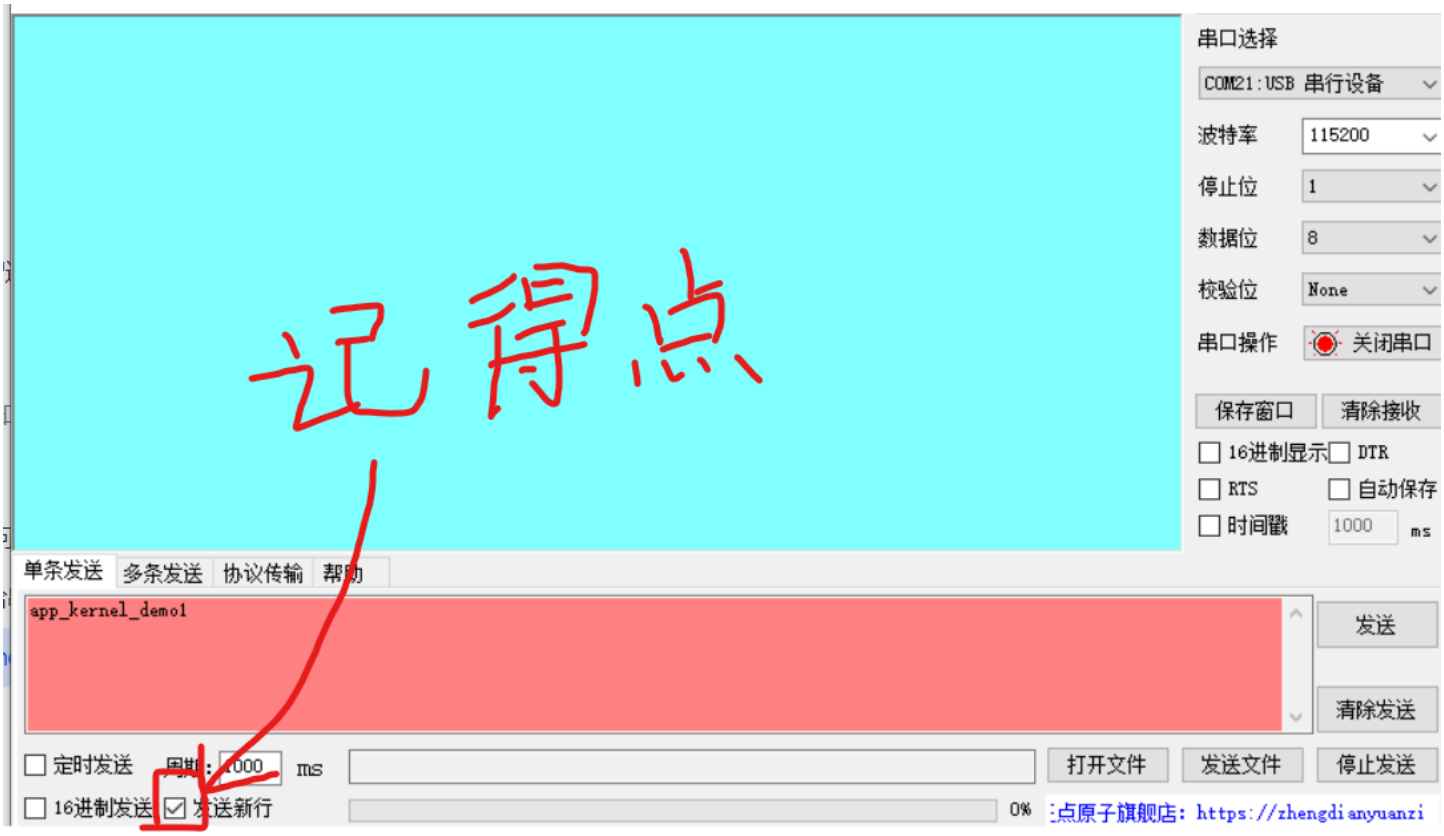
示例程序解读：

app_kernel为用户提供的api都有哪些？如何使用？看下面三个示例代码就够了。

示例程序总览以及如何调用示例程序：

一共有三个示例程序，app_kernel_demoX （X = 1,2,3），它们位于app_kernel.c文件的最后面。

调用它们，你可以打开app_kernel的串口（串口移植请看笔者的视频或.h文件注释），比如你想调用app_kernel_demo1，那你可以在串口助手输入app_kernel_demo1，再输入一个回车（串口助手一般都有自动换行功能，如果没有，那就直接按回车代替即可），然后发送给串口即可，点击发送之后，app_kernel会通过下文将要介绍的“信号与槽”机制自动调用示例程序app_kernel_demo1这个函数，并且将调用这个函数产生的日志信息打印到串口助手上。以正点原子的串口助手为例子：



app_kernel_demo1：

time：这个组件类似于闹钟功能，给它一个时间的参数（单位是系统节拍ticks）和一个回调函数指针，则将在当前时刻之后的第ticks那个时刻调用指定的回调函数。

call：这个组件专门用于和用户交互，当用户在串口助手里面输入“call name”，则会调用指定的回调函数，其中name和回调函数的绑定需要使用app_kernel_regist_user_call_function注册，用户在串口助手中里输入call list可以查看当前系统中已经注册的所有的name。

```
/*******************************************************************************
函数功能：call和time示例
入口参数：略
返回  值：无
*******************************************************************************/
//time命令回调函数
void time_testFunction(void* param,uint32_t argc){
        APP_KERNEL_LOG("%s\r\n", (char*)param);
        vTaskDelay(200);
}

//call命令回调函数
void call_testFunction(char*param[],uint32_t argc){
        APP_KERNEL_LOG("Hello world!\r\n");
```

```
        for(int i = 0; i<argc; i++){
                APP_KERNEL_LOG("%7.2lf ",strtod(param[i], NULL));
        }
        APP_KERNEL_LOG("\r\n");
}

void app_kernel_demo1(void* param, uint32_t argc){
        APP_KERNEL_LOG("\r\napp_kernal: run app_kernel_demo1...\r\n");
        APP_KERNEL_LOG("app_kernal: 正在运行call和time服务示例程序...\r\n");

        //以下是time服务的应用场景，分别在500，1000，1500各系统节拍之后调用time_testFunction
  //函数打印不同的参数
        char time1_param[] = "time1 test";
        char time2_param[] = "time2 test";
        char time3_param[] = "time3 test";
        app_kernel_call_after_times("time_test1", time_testFunction, time1_param, strlen(time1_param)+1, 1500);
        app_kernel_call_after_times("time_test2", time_testFunction, time2_param, strlen(time2_param)+1, 1000);
        app_kernel_call_after_times("time_test3", time_testFunction, time3_param, strlen(time3_param)+1, 500);
        app_kernel_show_times();

        //注册一个call服务，名字为call_test，在串口终端输入"call call_test"再输
        //入回车即可调用call_testFunction函数
        app_kernel_regist_user_call_function("call_test", call_testFunction);

        //提示一下：有用户可能有疑问为什么没有提供这种接口：在串口终端调用某个指令，然后
        //延迟一段时间再执行指令指定的函数，其实这个需求只要结合call服务和time服务即可轻松实现
}
```

运行效果：

```
////////////////////////////////////////////////////////////////////////////////
app_kernel: 用户发送的原始命令: app_kernel_demo1


app_kernal: run app_kernel_demo1...
app_kernal: 正在运行call和time服务示例程序...
app_kernel: 当前内核中含有的time服务有以下几个:          (按照调用时间的先后顺序排列)
time_test3->time_test2->time_test1
time3 test
time2 test
time1 test
////////////////////////////////////////////////////////////////////////////////
app_kernel: 用户发送的原始命令: call list

app_kernel: 当前系统中注册的call服务有以下几个:
call_test
////////////////////////////////////////////////////////////////////////////////
app_kernel: 用户发送的原始命令: call call_test 1 2 3

Hello world!
   1.00    2.00    3.00
```

time服务提供了比软件定时器更加易用的方案，考虑这样一个需求，当触发某个事件之后，过一段时间再执行某些操作。time专为这种需求而生！只不过它不允许在中断中调用，不过后续示例程序的signal可以在中断中使用，结合signal和time照样可以实现上述需求.

call服务和用户的关系最为亲密，因为它直接和串口助手交互，之前我做过一个飞控的项目，需要调节pid参数，其实就是改变程序中某一两个变量的值，但是每次改变这些变量都重新再烧录程序太麻烦了，也废单片机的flash，所以我就考虑用call来解决这个问题，直接通过call在串口助手中给call的回调函数传入我要改的参数，当然参数默认是字符串类型，我实际需要的是浮点类型，借助c语言的库函数strtod即可把字符串转为浮点数.轻松解决上述需求！这样不需要每次重新烧录程序，只需运行一次程序，然后一只借助串口助手来非常快速的调节参数即可！

再比如，我想靠串口助手来让飞控的电机锁定（即强制停转）那我只需要注册一个call，然后在他的回调函数里面写电机停转的逻辑，通过串口锁定电机即可，而无需手动的提心吊胆的断电

还有一个例子，之前实验室给一个学妹布置了一个单片机入门的学习项目：通过串口助手的指令去点灯，即用户通过串口发送某个指令，那么红灯亮，发送另外一个指令那么篮灯亮，发送第三个指令，那么绿灯亮。这个例子可以直接使用三个call来实现：比如注册一个call，它的name参数是red，然后

在它的回调中直接点亮红灯，这样的话，当用户用串口助手发送call red，红灯就会亮；其他两个灯也类似

说来惭愧，当时我指导学妹的，时间太紧，没来得及和学妹分享这个方法。并且我当时用一般的串口协议解析的实现的方法，也没时间和学妹讲清楚代码

app_kernel_demo2：

这个功能就是在背景篇提到的“信号与槽”的例子，定义一个signal_t 类型的信号对象，然后通过connect为此信号绑定一个槽函数，则之后可以在这个工程的任意任务，任意中断（中断必须是freertos可管理的中断）中通过调用信号发送函数（在任务中是emit，在中断中是emit_FromISR）来通知内核线程在合适的时机调用槽函数，并且槽函数的上下文是任务上下文，而不是中断上下文，也就意味着你可以在槽函数中延时！

信号与槽这个机制，我着重强调一下，我是先开发完一个叫signal的组件，它完成了与Qt中的信号与槽相同的功能，只不过api的接口名字不同而已，在使用上有极小的差距。在此时我是不知道有“信号与槽”这个东西的；之后去了解了qt，结果发现qt的“信号与槽”机制就是我这里signal组件实现的功能！（嘻嘻嘻，浅浅的装一波！）

虽然这个功能是我独立创造出来的，但是人家Qt的制作者比我更提前的创造出来了它，所以我就将开发好的signal组件的api名字改变，和Qt保持一样，这样也有利于熟悉Qt的读者使用app_kernel。不过在开发日志里面可能还是会出现signal组件，读者需要知道他其实就是这里的“信号与槽”。不改变开发日志的名字也是为了向幼年内测版本的app_kernel致敬。

```
/*************************************************************************
函数功能：信号与槽示例
入口参数：略
返回  值：无
*************************************************************************/
/*定义槽函数*/
void print(void* param, uint32_t argc){
        APP_KERNEL_LOG("param : %s, argc : %d\r\n", (char*)param, argc);
        vTaskDelay(21);
}

/*信号一般被定义成全局变量，这是因为信号主要是用来同步任务与任务，任务与中断的，
 所以他需要被不同的任务，中断访问到，因此要定义成全局变量，对于一些简单的的应用场
 合定义成局部变量当然也是可以的*/
signal_t signalTest;
uint8_t flag = 0;
```

```
void app_kernel_demo2(void* param, uint32_t argc){
        /*初始化定义好的全局变量signalTest，但是用户可能多次调用用例2
          多次调用不应该重复初始化信号，因此这里加入flag判断，只允许初始化一次*/
        if(flag == 0){
                signalTest = Signal();
                flag = 1;
        }


        /*连接信号与槽*/
        connect(signalTest, print);

        /* 发送信号十次，并传入参数Hello world，这里一定要注意emit的
           第三个参数argc一定要是第二个参数指向的内存大小（单位字节），
           如果argc过小，则给槽函数传递的param是不完整的，如果argc过大则
           会因访问非法空间直接导致系统崩溃*/
        for (int i = 0; i<100; i++){
                emit(signalTest, "Hello world!", strlen("Hello world!")+1);
                vTaskDelay(20);
        }
        /*删除信号，这个函数慎用，信号一旦删除，如果还试图发送信号，将导致系统崩溃*/
//        del_signal(signalTest);
}

```

运行效果：

```
////////////////////////////////////////////////////////////////////////////////
app_kernel: 用户发送的原始命令: app_kernel_demo2

param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13

param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13

param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
param : Hello world!, argc : 13
```

“信号与槽”机制以最直接的方式的实现了rtos的终极目的：同步任务与任务，任务与中断。

任何任务和中断都可以分别通过信号发送函数emit，emit_FromISR来通知app_kernel执行信号对应的槽函数，槽函数是任务上下文，这就给用户提供了极大便利，可以大胆的延时，调用其他rtos服务。

更重要的是，俩个信号发送函数emit，emit_FromISR都是非阻塞的，被调用过程非常迅速，不会对调用任务产生时间上的影响，这就是我前文提到过的异步机制的优势之处。

app_kernel_demo3：

非阻塞延时，专用于状态机编程，首先定义一个延时对象ak_nonBlockDelay_t ，然后通过app_kernel_non_blocking_delay_reset给这个延时对象指定一个时间，则只有系统时间到达当前时

间加上刚刚指定的时间以及之后，调用app_kernel_non_blocking_delay函数才会返回1，否则一直返回0，通过这两个函数配合，就可以完成状态机的指定时间非阻塞状态转换。代替了传统的计数器实现方案。

实际上，有了rtos之后，像一些简单的按键检测，芯片超市等待等场景更推荐使用rtos提供的延时函数，而无需担心裸机代码那样死等待浪费cpu，因为操作系统发现这个任务延时了，任务调度器会去处理其他就绪态的任务，不会像裸机那样死等待浪费cpu的！这也是rtos的一大魅力。rtos的延时函数在某些场景简单方便比这里的非阻塞延时更加易用。

但是，有的时候，这里的非阻塞延时是无法用rtos的延时替代的，他们俩个最大的区别是：rtos的延时函数无法重复的执行某个任务的同一段代码，延时就是延时了，这个任务直接就被阻塞了，什么也干不了了！而这里的非阻塞延时（包括我提到的计数器延时）只是查询一下时间状态，如果时间未到，可以去干这个任务里面其他到事情！就比如这个示例代码4，我在状态1的非阻塞延时期间可以持续打印字符串“state1”，只有时间到了之后，才会转状态2。这是普通的延时函数无法做到的！

```
void app_kernel_demo4(void* param, uint32_t argc){
    uint8_t state = 1;
    vTaskDelay(50);

    /*定义非阻塞延时对象(ak_nonBlockDelay_t类型)*/
    ak_nonBlockDelay_t testDelay;

    /*复位ak_nonBlockDelay_t对象，重新开始计时，在当前时间1000个ticks之
      后，调用app_kernel_non_blocking_delay函数传入testDelay对象将返回1*/
    app_kernel_non_blocking_delay_reset(&testDelay, 1000);
    while(1){
        switch (state){
            case 1:
                APP_KERNEL_LOG("state1\r\n");
                vTaskDelay(200);

        /*判断testDelay对象是否到达指定时间，如果未到达返回0，到达了返回1，此函数时非阻塞的*/
                if(app_kernel_non_blocking_delay(&testDelay)){
                    /*复位testDelay对象，重新开始计时*/
                    app_kernel_non_blocking_delay_reset(&testDelay, 1000);
                    /*状态转换*/
                    state = 2;
                }
                break;
            case 2:
                APP_KERNEL_LOG("state2\r\n");
```

```
                vTaskDelay(200);
                if(app_kernel_non_blocking_delay(&testDelay)){
                    app_kernel_non_blocking_delay_reset(&testDelay, 1000);
                    state = 3;
                }
              break;
            case 3:
                APP_KERNEL_LOG("state3\r\n");
                vTaskDelay(200);
                if(app_kernel_non_blocking_delay(&testDelay)){
                    app_kernel_non_blocking_delay_reset(&testDelay, 1000);
                    state = 4;
                }
              break;
            case 4:
                APP_KERNEL_LOG("state4\r\n");
                vTaskDelay(200);
                if(app_kernel_non_blocking_delay(&testDelay)){
                    APP_KERNEL_LOG("用例4测试结束\r\n");
                    return ;
                }
              break;
        }
        vTaskDelay(1);
    }
}
```

运行效果：

```
////////////////////////////////////////////////////////////////////////////////
app_kernel: 用户发送的原始命令: app_kernel_demo4

app_kernal: 调用信号app_kernel_demo4的回调函数!
state1
state1
state1
state1
state1
state2
state2
state2
state2
state2
state3
state3
state3
state3
state3
state4
state4
state4
state4
state4
用例4测试结束
```

状态机编程非常重要，适应范围很大！裸机，rtos的应用程序都经常看到它的身影，它直接和单片机应用程序相关！

这里注意我指的是应用程序，而不是驱动程序！比如你写一个电机相关的程序，那么怎么让电机以一定速度和方向转起来属于驱动程序，而怎么利用写好的驱动程序，去让电机在合适的时间转起来，遇到某个事件让电机调速，这些属于应用程序。简单来说，驱动程序负责提供一系列函数驱动具体的硬件正常工作，而应用程序负责使用这些驱动程序去做用户想做的事情，负责直接与单片机的业务逻辑打交道。

之前我有一个fpga的专业课，我就发现，fpga的vhdl应用程序也都是状态机！你能想象，大几千行代码，仅仅只是一个状态机，这当时对年幼的我造成了多么大的冲击！它之所以这么重要，是因为C语言，乃至单片机的应用程序本质上还是面向过程的，什么时间该干什么事情，或者触发了什么事件该干什么事情。这就注定了状态机与单片机是最佳搭档！

引入rtos之后，我们的编程虽然变简单了，对状态机的依赖程度有所降低，但是状态机仍然是编写单片机应用程序的一大法宝！一切花里胡哨的应用需求，当使用状态机的思想来编程，都会变得那么简单。可以这么说：面向过程的尽头是状态机编程。

开发日志：

开发日志是记录开发本模块的过程中遇到bug，解决bug的过程，以及开发新特性的地方，类似于日记的作用，所以不建议读者看，因为内容可能过于枯燥无味.

1.如何移植，移植好之后是什么效果，可以参考app_kernal.c开头的注释，但是刚建议看我的视频手把手教学

2.使用简介-看三个示例程序

增加了对app_kernel模块的认知，其可管理内存必须在6000字节以上才能保证系统正常运行（再编辑于2025/1/21：最新内测版本的可管理内存在20000字节，这个量的最低要求待测试），且涉及到外设中断的优先级不能太高，抢占优先级5是临界点，比它优先级高的中断中禁止调用freertos api，否则会导致系统卡死；开发app_kernel模块的延时调用用户函数功能（闹钟功能，相比于单次模式的定时器，调用更简洁，且内置在app_kernel模块中)；

1.对app_kernel模块添加新功能-信号机制，他的主要函数及解释如下：

APP_KERNEL_LOG("signal服务是一系列signal函数的集合，包括：\r\n");

APP_KERNEL_LOG("app_kernel_signal_callback_enroll　：　在内核中注册信号\r\n");

APP_KERNEL_LOG("app_kernel_send_signal　：　发送信号请求内核处理回调函数（任务上下文）\r\n");

APP_KERNEL_LOG("app_kernel_send_signal_FromISR　：　发送信号请求内核处理回调函数（中断上下文）\r\n");

APP_KERNEL_LOG("app_kernel_del_signal　：　删除信号\r\n");

APP_KERNEL_LOG("以上就是所有的signal函数集，了解signal的详细用法，请阅读以上几个函数的注释.\r\n");

实际使用的话，首先要调用app_kernel_signal_callback_enroll函数绑定一个回调函数，返回一个ak_signal_t类型的信号对象，此时信号服务已经成功注册，之后在任务上下文的任何地方调用app_kernel_send_signal即可触发回调函数调用；同时还支持在中断上下文触发回调函数调用，使用app_kernel_send_signal_FromISR函数即可满足这个需求。

2.优化了上述的信号机制，因为实测发现当多测调用信号服务的话会导致系统崩溃，经过分析得出bug的原因：因为每次发送信号都会调用任务创建函数，会产生内存消耗，虽然说可以通过及时删除信号来缓解系统卡死的问题但是不能根治，原因是不断的申请内存，释放内存会产生大量内存碎片，当积累到一定程度，系统已经无法申请到足够大的内存块了，此时如果再次试图申请内存就会导致系统卡死，使用即使删除信号并不能根治卡死的现象。经过思考，通过freertos的队列机制解决这个问题，通过链表存储当前系统app_kernel_signal_callback_enroll 注册的所有回调函数，然后只使用一个任务来在合适的时机调用这些函数，这样就无需像之前那样大量申请内存释放内存了，空间复杂度由O(n)变为了O(1)；问题成功解决，现在的app_kernel模块只要成功注册信号，就可以无限次通过发送信号来调用回调函数！

3.优化了app_kernel的其他两个服务call和time，之前这两个服务注册相关的逻辑不受os内核的保护，在使用的时候很可能因为被其他任务打断而产生对同一个链表的竞争访问行为，这是很大的一个隐患（算然目前为止并没有发生这个隐患）；解决这个隐患的办法是每次进行某些公共资源访问的时候直接挂起调度器禁止任务打断

发现app_kernel的一个小bug，当使用call功能连续注册相同的call服务，系统仍然可以成功注册，这是不太合理的，系统中不应该允许同名的call服务，因为这可能引起未知的冲突，并且重名服务本身也是不合理的，因此优化了call服务的逻辑：现在如果已经成功注册了一个call服务，当用户再次试图注册相同名字的call服务时，系统会提示错误：系统中已经注册了call服务：%s，无需重复中注册！

通过下面我在工作中遇到的案例，要对任务栈特别注意！：

周四.周三最后编写的代码经过实测发现巨大bug，dtu上行数据莫名其妙的不会被单片机接收，为了解决这个bug，首先，我放弃使用dtu，直接用电脑的串口助手向上位机发送任意数据，实测发现单片机可以正常接收数据，而使用dtu发送特定的命令数据系统就会崩溃，此时我怀疑是dtu的特定命令数据导致的bug，然后用串口助手连接dtu，让其发送数据，然后将这些特定命令数据记录下来，然后再次将串口助手连接上单片机，发送刚刚dtu发送的特定数据，结果发现确实是这些特定的命令数据导致的系统崩溃，印证了我之前的猜想：产生bug的原因确实是dtu特定命令数据导致，并不是单片机串口接收有问题。接下来的任务就是通过串口助手来准确的产生这个bug（为什么不直接用dtu？这是因为，dtu需要通过手机才能发送指令，用串口助手更加的方便调试，可以灵活的复现这个bug）

但是此时能灵活的随时产生这个bug只是解决这个bug的起点，之后我进行过多次调试，发现运行任务的每一个与dtu相关的函数都有可能导致系统崩溃，这就很难定位错误究竟出现在了哪里，我尝试过各种调试方法，结果是多个地方都有可能导致系统崩溃，而改变其他地方的代码，某个地方又不会导致系统崩溃，此时我就猜测，大概率产生这个bug的原因并不是代码逻辑写的有问题导致的，而可能与代码执行过程有关系，执行过程越少，越不容易产生bug，为什么我是这么"猜"的，因为前面已经说过了，当我用串口助手发送无意义的指令，这个时候程序不需要响应这个指令，那么执行过程自然就少了，此时并没有导致系统崩溃，而当我发送有意义的指令，则程序需要响应，响应的过程需要执行很多代码， 此时系统就会崩溃，据此，得出结论，程序崩溃的原因是：程序执行过多指令了？啊？是这

个原因吗？是的，确实是，但是更进一步想，为什么执行过多指令就会崩溃？我猜测，执行过多指令会消耗更多的栈空间，因为程序是在任务站里面运行，任务栈的空间是有限的，用完了，如果还试图定义局部变量，系统就会崩溃，因此，把任务栈的空间调大一点即可，我将任务栈的空间从原来的128bytes改为512bytes，然后做实验，bug解决！

1.发现ak模块的一个bug，time服务的传参行为是址传递，而与之相关的用户api都是非阻塞式的，因此可能发生传递的参数所指向的内存地址失效的情况，如：在一个函数里面定义好参数，然后将其指针传递给time函数的api，结果这个函数立刻退出，此时指针指向的地址是无效的（除非通过static或者申请内存函数将其定义在堆区或静态变量区），这可能导致不确定的行为发生，因此今天改进了这个bug，改进方法是将址传递改为值传递，由于time服务需要传递的参数是指针类型，因此只要保证一开始就为其分配足够的内存，然后参数传递的话，将要传递的参数值拷贝到这些内存地址，这样就可以避免上述bug可能导致的非法访问。但是程序是如何知道该为time服务的参数分配多大内存呢？为此，我定义了另外一个参数argc，它指定了要为参数param（泛型指针）分配多大内存空间

2.发现ak模块另外一个可能的bug，ak内核对象：time，call，signal等数据结构，他们通常都有一个指向字符串的成员-name，之前我直接将其定义为字符指针类型了，但是现在细想不太合理，它只是一个指针，并没有指向实际的内存空间，那万一实际的内存空间被随时释放了，程序就有可能出错，虽然到目前为止还没发现类似的错误发生！为什么现在不会出错，因为在程序里面，一般我是这么给它赋值的，以time为例子：

time->name = "head";

上述例子中，等价表示为：

char* name = "head";

根据这个等价表示的语法，我查阅相关资料得知，此时"head"这个字符串它被定义在"常量区"，这个区域的内存不会因为函数退出而被操作系统清除，因此这就是前文提到的，这个潜在”bug“从来还没有触发过的原因，但是从严谨性上考虑，即使不会发生这个bug，那这样写代码也是不合理的，不应该让指针指向一个在函数内定义的"局部变量"（虽然它不是，但是类似），即使现在没发生这个bug，但是也仅仅是因为这个在函数内部定义的变量被定义在常量区了。

因此我改进了time，call， signal服务的name传递机制，改进措施和2.类似，即在内核对象初始化的时候就直接将name成员指针指向一块提前申请好的内存空间，然后将用户传递的name直接拷贝到这块内存空间即可

3.解决了存在歧义的一个点：ak模块的示例代码3“app_kernel_demo3”其最后两行

app_kernel_del_signal(signal1);

app_kernel_del_signal(signal2);

这两句过后，两个信号从app_kernel内核中删除，并清理内存，这里注意运行效果是先将app_kernel_demo3整个函数运行完毕，再开始执行signal1和signal2的回调函数，因为app_kernel_demo3也是通过signal调用的（用户在串口终端输命令调用的情况是这样）而signal服务的回调顺序是fifo顺序，即先进先出。

因此这里用户可能会担心：app_kernel_demo3运行完毕了，则其最后两行将两个信号都删除了，此时信号在系统中已经不存在了，那么系统还能执行不存在的信号signal1和signal2的回调函数吗？其实是可以的，信号发送函数会将信号对象拷贝一下然后传递给app_kernel内核，处理回调函数的任务，所以说，在调用完信号发送函数的那一个时刻，app_kernel模块就一定能保证成功调用到其回调函数！因此无需担心上述问题！这样的机制提高了signal服务的鲁棒性！

2025/1/17

今天接触了了qt，我意识到，原来我在app_kernel模块中实现的被我自称做“信号”的这个东东，原来在qt中也有类似的实现，qt中这个机制叫做”信号与槽“，它的简介：Qt 框架的核心特性之一，它提供了一种非常灵活和强大的对象间通信机制。原来在我设计app_kernal的同时竟然能和QT的核心特性产生联系！不过最初我命名app_kernel的信号，是因为之前学习过linux系统应用编程，里面也有信号的概念，我借鉴了linux的思想，直到今天才知道，原来qt中也用到了这种思想，并且个人认为，qt里面”信号与槽“的和我在app_kernel里面实现的“信号”它们在使用上非常类似！

新增非阻塞延时函数-app_kernel_non_blocking_delay，此函数笔者初见于px4无人机厂商写的一个功能包，它解决了我们日常写状态机的时候，需要持续一段时间执行相同的一些操作，类似于延时，但是又不太一样，因为它是不断地执行相同的代码块持续一段时间再转状态的，其实没有这个函数，我们照样可以通过状态机代码的执行周期来通过计数器变量达到同样的目的，但是如上文所说，这样的代码就不够美观了，也不好理解，并且计数器也经常容易忘记清零，或者忘记赋初始值等导致的bug（可能是我比较菜吧，反正我用计数器写状态机确实容易出现这些问题，不过写多了习惯之后就不会出现了）。为此，我特意写了一个app_kernel_non_blocking_delay，以及这个函数的状态机的测试用例-app_kernel_demo4，对比一下它的核心代码和使用计数器实现的状态机代码，就可以发现它还是比较简洁的，但是不一定好理解，理解的前提是知道app_kernel_non_blocking_delay的工作机制，不过也不算难理解，中规中矩吧，可能有些人觉得计数器的实现方式更加直观，这我也无法反驳...

实际上，有了操作系统之后，像一些简单的按键检测，芯片超市等待等场景更推荐使用操作系统提供的延时函数，因为简单方便比状态机更加易用。不过，这里的非阻塞延时函数也是有应用场景的

2025/1/19

临近过年，打算考虑彻底增强signal机制到最终发布形态，signal目前一共经历过两个大版本迭代（在开发日志中有所体现）：第一个版本：初版存在很大缺陷，无论发送同一个信号还是不同信号，都会重新创建任务，调用玩函数又会删除任务，这个机制产生的内存碎片是单片机无法承受的，实测发现信号发送次数超过十次以上系统就会崩溃；第二个版本：二代版本也就是目前的版本（指2025/1/19）改善了上述问题，采用有一个任务全局调度所有信号的回调函数，调度机制是fifo及队列化调度.这次确实彻底解决了初代版本存在内存碎片的问题，因为这个版本自始至终都只有一个任务，但是代价是信号的实时性相较于初代版本被削弱了，因为采用根据发送信号时间队列化调用回调函数的机制，导致有一些很紧急的信号由于发送时机较晚而迟迟得不到执行，这也说明我的第二代设计实际上还是不支持在回调函数中延时的，因为会影响其他的信号！

因此趁着今晚拉肚子加失眠，我连夜想出第三代signal实现的初步设想：既然第一代的缺点是浪费内存，但是优点是实时性较高，因为不同的信号甚至同一信号的不同发送周期都有独立任务，实时性可以说是非常之高。而第二代的缺点是实时性很低，这是信号回调函数调度任务的队列化管理机制导致的，但是优点很明显，只有一个回调函数处理任务，完全不浪费内存。

那么我可以采取一个折中的方案：还是像第一代那样子，为每个信号都申请一个回调任务，因为用户在系统里面用到的信号次数总归还是有限的，不会无限注册信号！但是与一代有所差别的是，对于同一个信号，我不会在发送信号的时候创建任务，回调函数执行完毕删除任务，不会这样做！而是仅仅在注册信号的时候创回调任务！因为前文提到过，用户注册的信号是有限的，所以这就可以避免一代那样优于信号无限发送而导致系统崩溃！这样做，每个信号还是像第一代那样有一个自己的回调函数，实时性很强；并且像第二代那样节省内存（虽然没有第二代节省，但是这个代价换来的是突飞猛进增长的实时性！），这样就对一二两代signal取长补短，达到了真正意义上的signal最终形态，可以与qt里面的“信号与槽”机制相媲美了！

time组件，其工作的过程中还是会产生内存碎片。这个问题一定要解决！可以考虑指定好系统当前可支持的最大time组件数量，当注册超过该上限，打印log错误信息，提示用户.

time组件，signal组件，call组件都应该有一个最大注册上限（待斟酌，有些确实没多大必要）

2025/1/20

对信号与槽机制的史诗级优化：

新思路：开三个线程，专门负责处理signal，也就是说，同一时刻仅仅允许同时处理三个信号（协程处理信号事件）

具体实现细节：发送信号函数需要判断当前哪个线程空闲，如果空闲，则通过事件机制通知对应线程处理此次信号回调函数，逻辑如下：

如果线程1空闲，则让线程1处理信号回调（问题是，线程1如何拿到需要处理的信号？）

如果线程1忙，则让线程2处理信号回调

如果线程1和线程2都忙，则让线程3处理回调

线程空闲事件应该都在等待事件组置位，事件组由发送信号函数置位，当线程处理完回调函数之后，需要将对应的事件位清零，然后等待下一次信号发送函数置位事件，进入新的循环

补充：这三个线程不要重新开了，而是使用app_kernal已经创建的三个线程，包括两个任务，一个软件定时器，其中软件定时器将作为线程3，这是因为，线程3一般很少用，而软件定时器3担任time组件的逻辑，工作量比较大

2025/1/21

临近过年，我迫切的想快点把这个项目发布出去！fighting!

首先我用旧版本暴力测试过signal的鲁棒性，果然，当发送信号被两个任务同时且频繁地发送的话，旧版本会立刻崩溃，坚持不过三秒，这更加坚定了我今天落实昨天想法的冲动

今天实现了昨天对信号机制进行优化的思想，并通过了暴力测试，新优化的信号机制，我将其接口封装成Qt风格的函数：Signal初始化信号，connect连接信号与槽，emit（包括中断版本emitFromISR)，诚然，虽然这个机制是我独立创造的，但是人家Qt的制作者确实比我更提前地实现了这个机制，所以就向标准化靠拢，沿用人家Qt风格的信号与槽api接口

今天做的另外一件事情，通过一个宏定义MAX_TIME_CNT来限制系统中同时可以存在的time个数，因为如果我不加这个限制，那么用户可能会非常频繁的调用time函数，而用户定义的time回调函数可能并不能处理的那么快，所以time的个数只增不减，任何一个计算机系统都无法忍受这种事情发生，因为没有无限大内存的计算机！这个措施进一步提高了app_kernel的鲁棒性！

至此，time，信号与槽机制（以后我就不叫它signal了，尊重Qt首创的这个机制！）在app_kernel上基本达到可以发布的状态了，至于call，用的并不算多，但是它是我写的app_kernel的第一个组件，已经在实践中证明过它的可靠性了，并且call服务无需暴力测试，这是因为用户的手速是有限的，不可能点串口助手点的那么频繁，明日再整理一下这个文档以及代码，打算正式发布了！

2025/1/21至1/22:

问题一：对信号发送函数应该给当前哪个线程发送信号的仲裁机制进行改进：线程正在处理信号的时候，此时消息队列已经接受完毕信号了，也就是说消息队列中有效消息个数此时已经减一了，而此时槽函数可能还未处理完毕…

问题2:极限状态下，三个线程消息队列都满了->此时都会陷入while循环处理回调函数->进而导致软件定时器停止工作->导致系统崩溃

优化以及改进措施：（1）再while循环加入os延时

（2）放弃使用timer线程，另开一个新线程，软件定时起器线程被阻塞会导致用例3异常，具体的 由于阻塞导致ak时基停止更新，导致状态机无法参照ak时基转换状态一直卡在状态1，2无法跳出

（3）问题2导致系统崩溃的原因，三个线程处理的槽函数有发送信号的语句，但是消息队列已经满了，这会阻塞三个线程，陷入死锁状态，一直被阻塞下去，无法处理消息队列。解决办法：发送信号函数设置为非阻塞形式，并对用户打印log：emit发送频率过快，导致丢失部分信号的警告信息

1/22上午上班过程空闲时间完成了上述所有设想，解决了所有测试用例可能产生的bug

除此之外，干了另外一件提升用户体验的事情，对app_kernel内核打印日志的宏APP_KERNEL_LOG加入互斥锁来保护它，这样就彻底杜绝串口中断上向用户反馈乱码的可能性，但是加入这个机制之后，发现time示例程序异常了，正在排查问题

经过排查，发现，time的回调函数中不能操作互斥量，一旦操作就会导致系统卡死，产生错误的原因是time的守护进程在任务调度器挂起期间，调用time回调函数，回调函数里面一旦进行延时或者操作互斥锁就导致的系统卡死；改进，在回调函数的上下文期间，开启调度器，允许其他任务抢占守护进程，但是这样又导致互斥量死锁产生：回调函数持有互斥锁，此时守护进程被抢占，然后高优先级的任务试图持有锁而产生死锁，进一步改进措施：放弃互斥锁保护机制，而采用简单粗暴的直接挂起调度器来保护log日志强制打印，问题圆满解决!

通过解决上述bug总结freertos的两个注意事项：

（1）在调度器挂起期间禁止延时和操作内核（指freertos内核）对象；

（2）操作互斥锁的时候，当任务持有锁的时候，要千万小心任务可能被高优先级的任务打断，并且高优先级的任务试图再次持有锁而陷入死锁状态。至于如何避免这个问题，除了管理好任务优先级，例如：高优先级的任务不要访问互斥锁，只能用相同优先级的任务访问，（这里读者可以想想为什么这样做不会产生死锁），笔者还没想到其他的避免措施，有想法的读者可以与我共同探讨一下。如果实在无法解决（2）我提出的问题，那么可以和我这里的做法一样，放弃使用互斥锁，直接挂起任务调度器来禁止高优先级任务抢占

中午吃饭之前的几分钟，我已经打算程序就定稿了，想要测试一下程序的最低内存要求，然后打印程序的剩余内存，结果突然想到time服务的内存碎片问题好像还没有解决。于是暴力测试time用例程序，发现内存碎片会导致系统崩溃。因此吃完饭后，趁着睡觉时间想出来并实现了解决这个问题的办法：杜绝每次调用time服务都要重新申请内存的行为，而是提前注册好系统支持的最大time的个数（以数组形式），当用户同时使用的time个数超过这个限制之后，还像之前那样打印警告信息，但是运行time的过程中绝不允许申请内存的行为发生，这就彻底杜绝了time产生内存碎片的可能

准备正式发布1.0稳定版之前，我测试了，当前系统运行所需的内存空间在10000字节左右，也就是说，freertos的宏定义configTOTAL_HEAP_SIZE的值至少要大于10000 bytes，才能保证四个测试用例能正常使用，实际使用建议设置为20000 bytes以上，因为用户还需要运行自己的应用程序，需要更多的内存。

最终版本的代码基本定型（时间节点：2025/1/21下午），目前的所有测试用例都能经受住各种暴力测试的考验，下一步将完善文档信息以及代码注释，然后就正式发布app_kernel1.0（稳定版）。