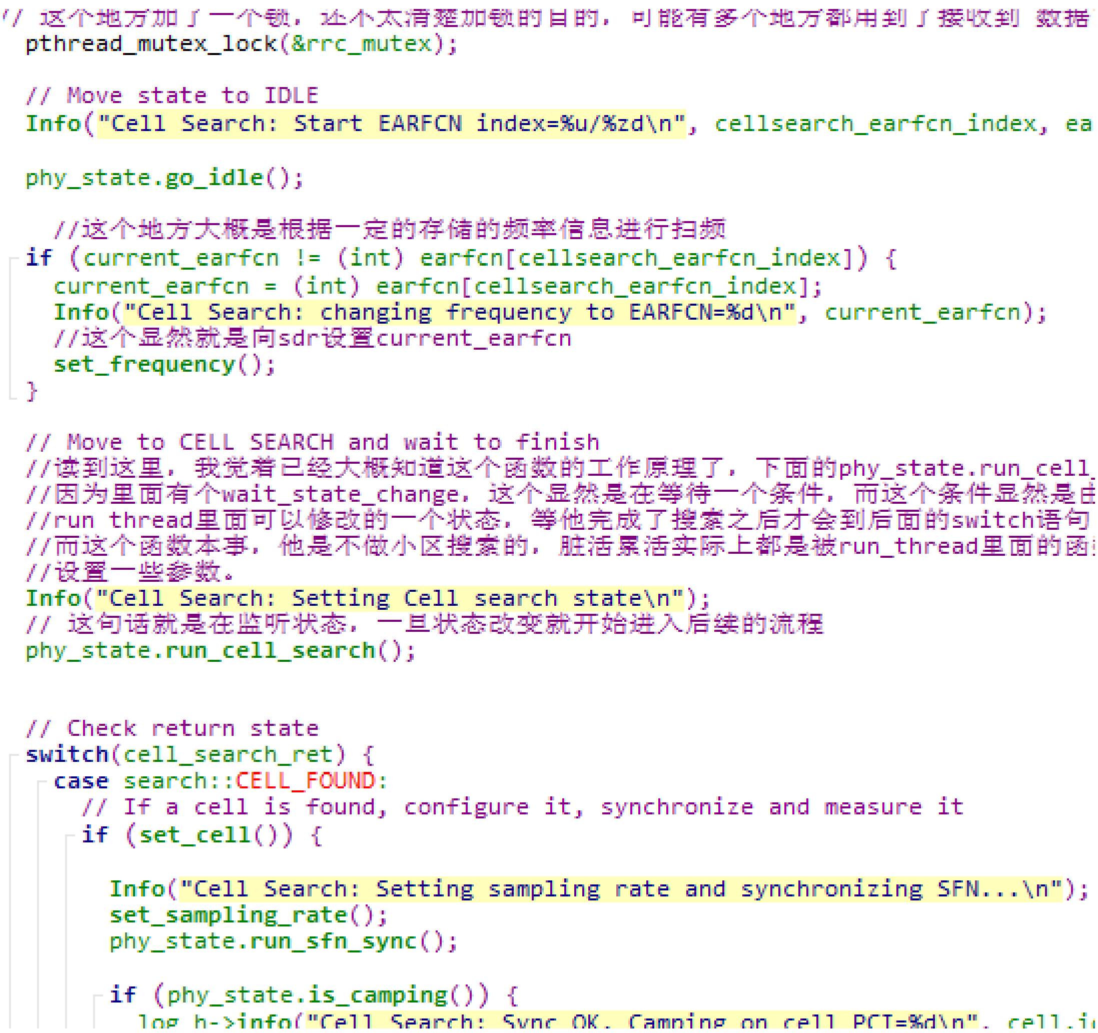
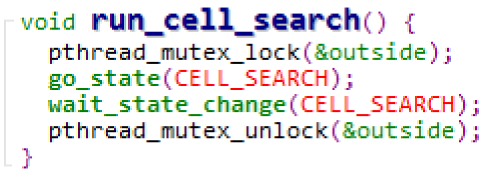
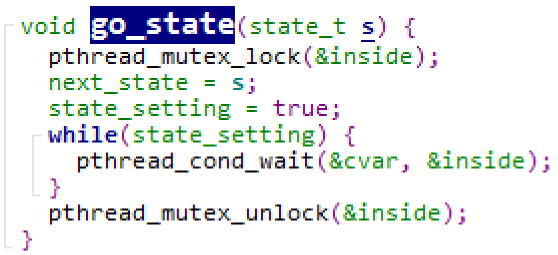


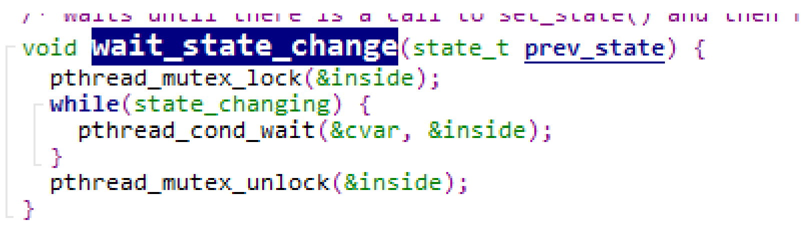
```
// 这个地方加了一个锁，还不太清楚加锁的目的，可能有多个地方都用到了接收到 数据
  pthread_mutex_lock(&rrc_mutex);

  // Move state to IDLE
  Info("Cell Search: Start EARFCN index=%u/%zd\n", cellsearch_earfcn_index, ea

  phy_state.go_idle();

    //这个地方大概是根据一定的存储的频率信息进行扫频
  if (current_earfcn != (int) earfcn[cellsearch_earfcn_index]) {
    current_earfcn = (int) earfcn[cellsearch_earfcn_index];
    Info("Cell Search: changing frequency to EARFCN=%d\n", current_earfcn);
    //这个显然就是向sdr设置current_earfcn
    set_frequency();
  }

  // Move to CELL SEARCH and wait to finish
  //读到这里，我觉着已经大概知道这个函数的工作原理了，下面的phy_state.run_cell_
  //因为里面有个wait_state_change，这个显然是在等待一个条件，而这个条件显然是由
  //run thread里面可以修改的一个状态，等他完成了搜索之后才会到后面的switch语句
  //而这个函数本事，他是不做小区搜索的，脏活累活实际上都是被run_thread里面的函
  //设置一些参数。
  Info("Cell Search: Setting Cell search state\n");
  // 这句话就是在监听状态，一旦状态改变就开始进入后续的流程
  phy_state.run_cell_search();


  // Check return state
  switch(cell_search_ret) {
    case search::CELL_FOUND:
      // If a cell is found, configure it, synchronize and measure it
      if (set_cell()) {

        Info("Cell Search: Setting sampling rate and synchronizing SFN...\n");
        set_sampling_rate();
        phy_state.run_sfn_sync();

        if (phy_state.is_camping()) {
          log_h->info("Cell Search: Sync OK. Camping on cell PCI=%d\n", cell.i
```



```
void run_cell_search() {
  pthread_mutex_lock(&outside);
  go_state(CELL_SEARCH);
  wait_state_change(CELL_SEARCH);
  pthread_mutex_unlock(&outside);
}
```



```
void go_state(state_t s) {
  pthread_mutex_lock(&inside);
  next_state = s;
  state_setting = true;
  while(state_setting) {
    pthread_cond_wait(&cvar, &inside);
  }
  pthread_mutex_unlock(&inside);
}
```

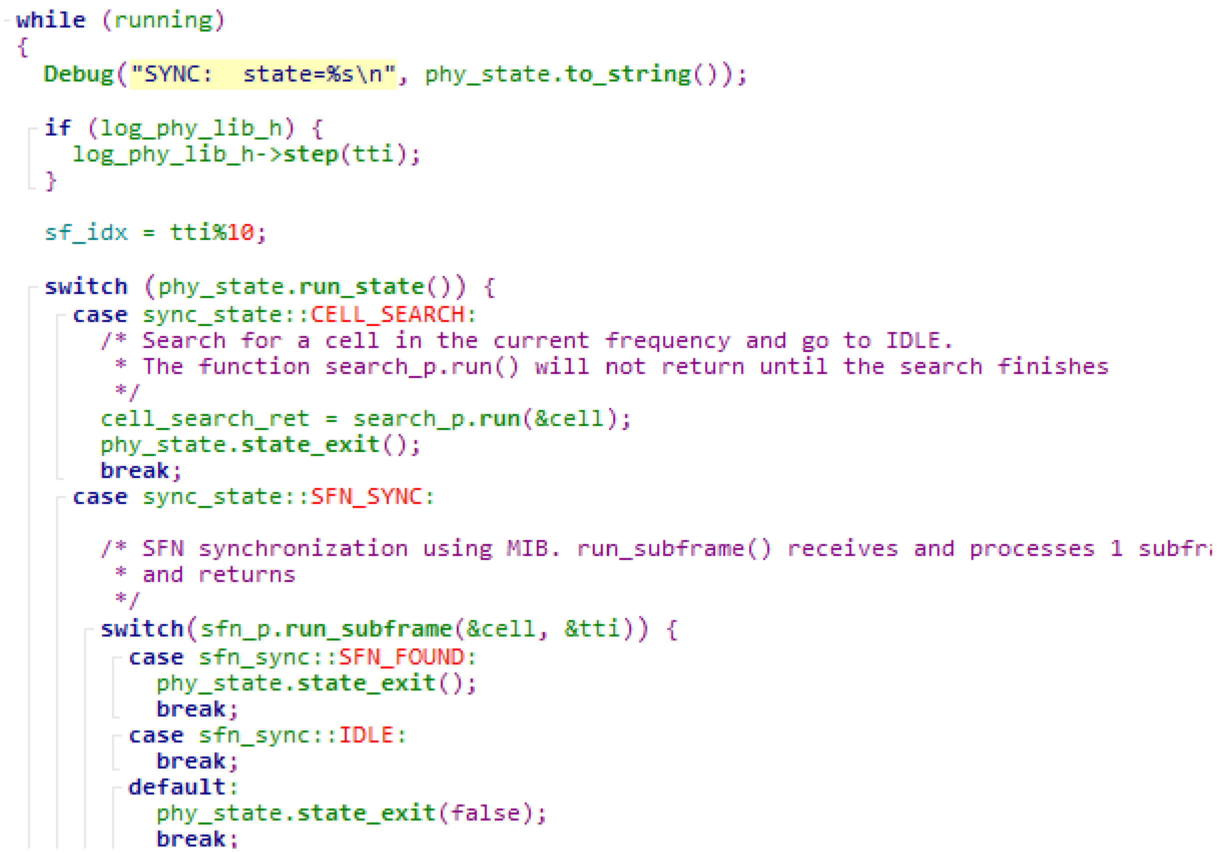


```
/* waits until there is a call to set_state() and then r
void wait_state_change(state_t prev_state) {
  pthread_mutex_lock(&inside);
  while(state_changing) {
    pthread_cond_wait(&cvar, &inside);
  }
  pthread_mutex_unlock(&inside);
}
```

根据之前的笔记，分析了为什么要加while循环，while循环可以自定义跳出条件，防止进群效应

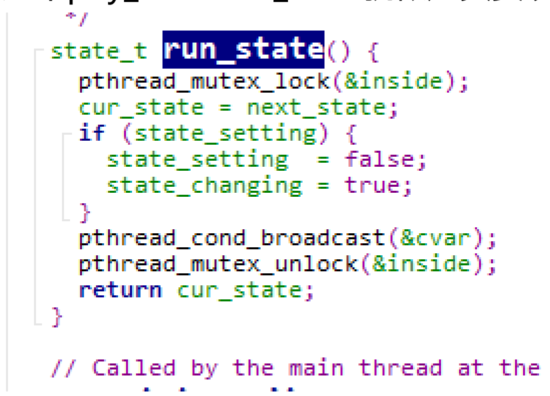
至于加锁不加锁，这些就太细节了，还有一种说法，在pthread_cond_wait里面传入的这个inside的mutex的目的是为了让state_changing可以被改变，而不是让cvar能够被改变

那我们就来分析一下唤醒方的代码是如何实现的

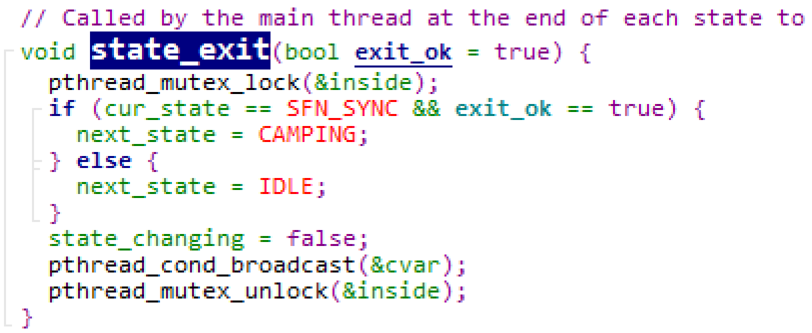


```
while (running)
{
  Debug("SYNC:  state=%s\n", phy_state.to_string());

  if (log_phy_lib_h) {
    log_phy_lib_h->step(tti);
  }

  sf_idx = tti%10;

  switch (phy_state.run_state()) {
    case sync_state::CELL_SEARCH:
      /* Search for a cell in the current frequency and go to IDLE.
       * The function search_p.run() will not return until the search finishes
       */
      cell_search_ret = search_p.run(&cell);
      phy_state.state_exit();
      break;
    case sync_state::SFN_SYNC:

      /* SFN synchronization using MIB. run_subframe() receives and processes 1 subfr
       * and returns
       */
      switch(sfn_p.run_subframe(&cell, &tti)) {
        case sfn_sync::SFN_FOUND:
          phy_state.state_exit();
          break;
        case sfn_sync::IDLE:
          break;
        default:
          phy_state.state_exit(false);
          break;
```

这个phy_state.run_state貌似是负责唤醒，更新状态的



```
 */
state_t run_state() {
  pthread_mutex_lock(&inside);
  cur_state = next_state;
  if (state_setting) {
    state_setting  = false;
    state_changing = true;
  }
  pthread_cond_broadcast(&cvar);
  pthread_mutex_unlock(&inside);
  return cur_state;
}

// Called by the main thread at the
```



```
// Called by the main thread at the end of each state to
void state_exit(bool exit_ok = true) {
  pthread_mutex_lock(&inside);
  if (cur_state == SFN_SYNC && exit_ok == true) {
    next_state = CAMPING;
  } else {
    next_state = IDLE;
  }
  state_changing = false;
  pthread_cond_broadcast(&cvar);
  pthread_mutex_unlock(&inside);
}
```

run_state的细节显示

他也是加了一个inside的所，然后去修改cur_state

接着去判断是state_setting 是否是true，如果是ture

再去修改state_changing

然后再去激活那个条件变量

那么问题来了，谁去修改了state_setting，为啥还要再来一个这？

谁又将state_changing改为了true

我们来理一下这个逻辑，首先run_thread会去读取nextstate将它赋值给 curr state

next_state的值又是在cell_search中的 run_cell_search中去实现的 ，

所以在cell_search可以去启动run thread的 search_p.run

因为run_cell_search是被两次阻塞的

分别被go_state阻塞了一次，而且需要强调的是 state_setting 是在go_state被置为true的

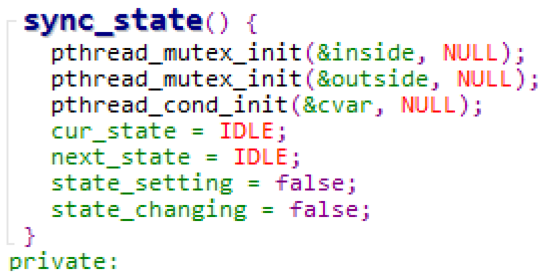
然后又被wait_change阻塞了一次

所以当run_thread的run_state执行的时候，go_state会被取消阻塞，进入到wait_state——change

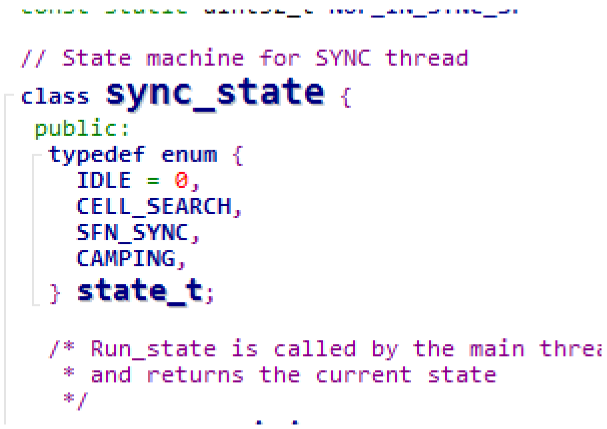
而后继续阻塞，当search_p run执行完了以后，执行完state_exit才会触发那个wait_state_change

之后，才会去执行cell search的其他内容

```
sync_state phy_state;
```



```
sync_state() {
  pthread_mutex_init(&inside, NULL);
  pthread_mutex_init(&outside, NULL);
  pthread_cond_init(&cvar, NULL);
  cur_state = IDLE;
  next_state = IDLE;
  state_setting = false;
  state_changing = false;
}
private:
```



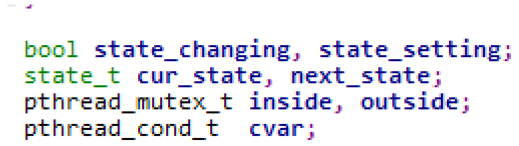
```
// State machine for SYNC thread
class sync_state {
 public:
  typedef enum {
    IDLE = 0,
    CELL_SEARCH,
    SFN_SYNC,
    CAMPING,
  } state_t;

  /* Run_state is called by the main threa
   * and returns the current state
   */
```

phy_state是一个sync_state的类型，构造函数自动初始化了cur_state和netx_state

物理层主要的就是维护phy_state这个状态机对象

这个状态机，的成员方法都是状态转移函数

而其成员变量的种类



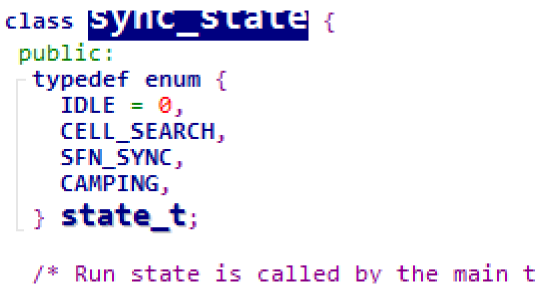
```
bool state_changing, state_setting;
state_t cur_state, next_state;
pthread_mutex_t inside, outside;
pthread_cond_t  cvar;
```

state_changing state setting 都是为了去更好的同步线程所用到的变量

cur_state next_state是两个状态机变量，至于为什么是两个我还不太清楚

pthread_mutex_t inside, outside这两个变量是锁

pthread_cond_t cvar 是一个条件触发的线程信号变量



```
class sync_state {
 public:
  typedef enum {
    IDLE = 0,
    CELL_SEARCH,
    SFN_SYNC,
    CAMPING,
  } state_t;

  /* Run state is called by the main t
```

这是状态机的几个取值

1. idle：接收数据，其他不干

2. cell_search： 第一次搜小区的时候用到

3. sfn_sync：

4. camping