**WVDS 前端设备自主开发**

**详细设计报告**

**无线组网**

WVDS是深圳市凯达尔科技实业有限公司的商标。

编制：研发中心--硬件部

版本：V1.0

项目名称：自主地磁开发

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **日期** | **版本** | **编制** | **审核** | **批准** | **发行纪录** |
| 2016年8月10日 | V1.0 |  |  |  |  |
| 2017年5月7日 | V2.0 | 陈亮 |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**目录**

[一. 开发/验证工具链 4](#_Toc32459)

[1.1. IAR环境Contiki开发 4](#_Toc20636)

[1.2. Cooja环境协议仿真 4](#_Toc30539)

[1.3. Sniffer协议分析程序 5](#_Toc5325)

[Sniffer节点程序及配套GUI程序用于观察实际节点网络运行时交互的无线数据包，从而实现对无线网络的问题诊断。 5](#_Toc5581)

[1.4. 演示后台系统程序 6](#_Toc7080)

[1.5. 完整开发环境搭建 7](#_Toc28466)

[1.6. 程序编译和烧录 8](#_Toc1586)

[二. 软件工程架构 10](#_Toc2441)

[2.1. App层 10](#_Toc28398)

[2.2. NWK层 10](#_Toc22322)

[2.3. MAC层 11](#_Toc24830)

[2.4. HAL层 11](#_Toc6142)

[三. 通讯协议 12](#_Toc24899)

[3.1. AP与服务器间通讯 12](#_Toc14425)

[3.1.1. 消息列表 12](#_Toc24921)

[3.1.2. 加密和编码 13](#_Toc18117)

[3.2. VD/RP与AP间通讯 14](#_Toc29609)

[3.2.1. 消息封装 14](#_Toc6705)

[四. OS架构与移植 15](#_Toc4334)

[4.1. process调度器 15](#_Toc22506)

[4.2. 时钟及timer 17](#_Toc721)

[4.3. 通信协议栈 17](#_Toc15930)

[4.4. 新节点平台移植 18](#_Toc25747)

[4.5. 射频驱动移植 18](#_Toc250)

[五. 底层驱动流程图 19](#_Toc29150)

[5.1. cc1120射频模块驱动 19](#_Toc12183)

[5.2. mmc3316磁传感器驱动 21](#_Toc23852)

[5.3. qmc5883磁传感器驱动 22](#_Toc6237)

[5.4. m26 GPRS模块驱动 22](#_Toc29292)

[5.5. m25pe存储器驱动 24](#_Toc31496)

[5.6. ds3231 RTC驱动 25](#_Toc12236)

[5.7. ADC驱动 25](#_Toc11494)

[六. 主函数流程图 25](#_Toc3537)

[6.1. 数据消息抽象 25](#_Toc8984)

[6.2. VD节点主程序 26](#_Toc14622)

[6.2.1. 磁场采样与算法处理 26](#_Toc15273)

[6.2.2. 关键数据消息存储 27](#_Toc21410)

[6.2.3. 外部Flash作为环形队列使用 29](#_Toc9368)

[6.2.4. 支持优先级的消息队列 29](#_Toc16388)

[6.3. RP节点主程序 30](#_Toc30733)

[6.4. AP节点主程序 31](#_Toc20858)

[6.4.1. 关键消息数据存储 33](#_Toc8011)

[6.4.2. 支持优先级的消息队列 33](#_Toc19325)

[6.4.3. 共享消息缓存和多消息队列 33](#_Toc23074)

[6.4.4. 关键数据作为历史数据发送 34](#_Toc14326)

[七. 各功能模块流程图和函数的输入输出说明 36](#_Toc30435)

[7.1. 信息存储模块 36](#_Toc4044)

[7.2. 节点激活模块 38](#_Toc26420)

[7.2.1. VD节点激活 38](#_Toc20740)

[7.2.2. AP和RP激活 40](#_Toc44)

[7.3. AT指令模块 40](#_Toc14039)

[7.4. 节点入网模块 41](#_Toc26784)

[7.4.1. AP节点入网 41](#_Toc11102)

[7.4.2. RP节点入网 43](#_Toc3720)

[7.4.3. VD节点入网 44](#_Toc1997)

[7.5. 心跳数据模块 45](#_Toc1254)

[7.6. Mesh网络模块 46](#_Toc8017)

[7.6.1. Mesh广播 46](#_Toc30107)

[7.7. 路由发现模块 47](#_Toc12811)

[7.8. 路由维护模块 47](#_Toc28416)

[7.9. 下行命令模块 48](#_Toc4291)

[7.9.1. 拦截式下行 50](#_Toc21319)

[7.10. 全网参数模块 50](#_Toc16193)

[7.11. 低功耗控制模块 51](#_Toc8178)

[7.12. 时间同步模块 54](#_Toc4724)

[7.13. 节点监测模块 55](#_Toc13787)

[7.14. 自检报警模块 55](#_Toc28880)

[7.15. 射频控制模块 56](#_Toc20131)

[7.16. 软件看门狗模块 56](#_Toc20100)

[7.17. 启动引导模块 58](#_Toc29319)

[7.18. 无线更新模块 59](#_Toc3191)

[7.18.1. 固件数据解析 59](#_Toc6635)

[7.18.2. 固件数据分发 60](#_Toc22582)

[八. 多任务交互 62](#_Toc16404)

[九. 软件仿真和测试用例及测试方法 64](#_Toc23599)

[9.1. Cooja软件仿真 64](#_Toc6381)

[9.2. 仿真分析方法 64](#_Toc19187)

[十. 附录一 65](#_Toc27089)

# 

# 开发/验证工具链

## IAR环境Contiki开发

本系统基于Thingsquare公司发布的thingsquare-mist-1.1.0.zip进行软件开发。Thingsquare公司由Contiki发明人Adam Dunkels博士创立并担任CEO，thingsquare mist是在Contiki操作系统基础上发布的软件包，提供了更多的硬件节点支持和AES加密等原Contiki系统当时未提供的应用特征。

本系统的开发针对凯达尔公司自主研发的硬件节点平台，MCU是MSP430F5438A、射频是CC1120加上功率放大器芯片CC1190。因此，软件开发时基于platform/mist-exp5438定义了适用于本节点的平台platform/cadre1120。

本系统的开发采用IAR Embedded Workbench for MSP430作为集成开发环境。开发团队内成员应采用一致的IAR EW430软件版本，避免不一致可能导致不同的编译错误。基本的IAR软件安装和使用不在本文档中赘述，可参见《移植教程》。

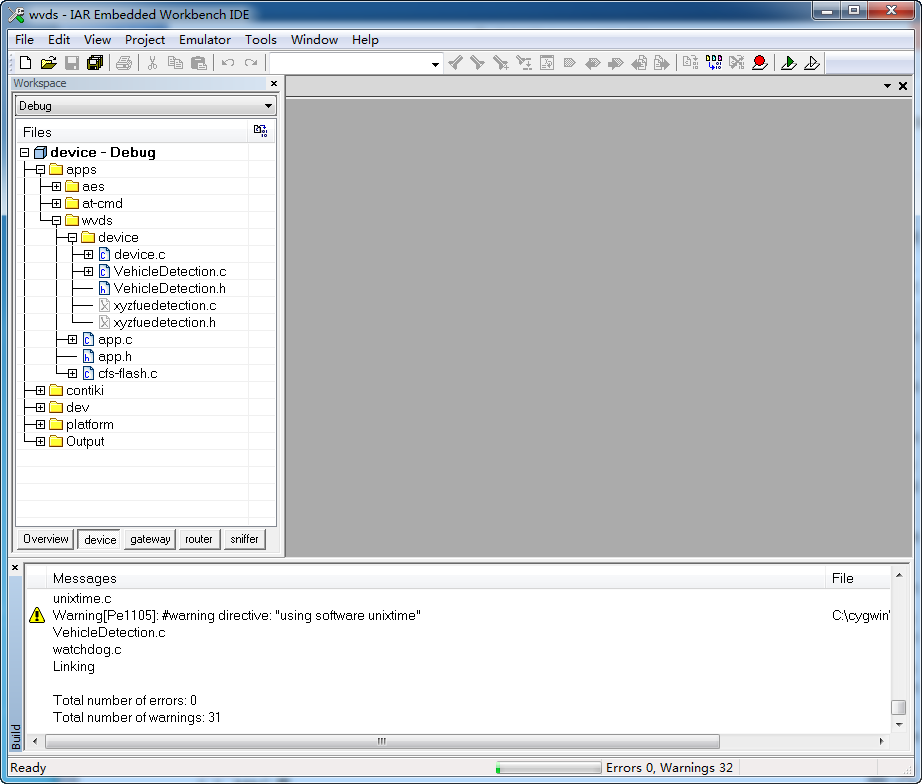


图 1 IAR下VD程序对应工程

## Cooja环境协议仿真

Cooja是Contiki操作系统自带的图形化网络仿真器。开发人员可以在图形化界面定义待仿真的网络拓扑，支持编译为几种真实的嵌入式节点程序进行仿真。通过Cooja环境下的软件仿真，可以更高效率地发现和修正网络协议的一些问题，特别是在节点较多的情况下。

Cooja仿真器的入门用法可参见文档 [https://mintisan.github.io/notes/get-started-with- contiki.html](https://mintisan.github.io/notes/get-started-with-contiki.html)。本系统涉及的仿真运行方法和仿真结果分析方法见第九节。

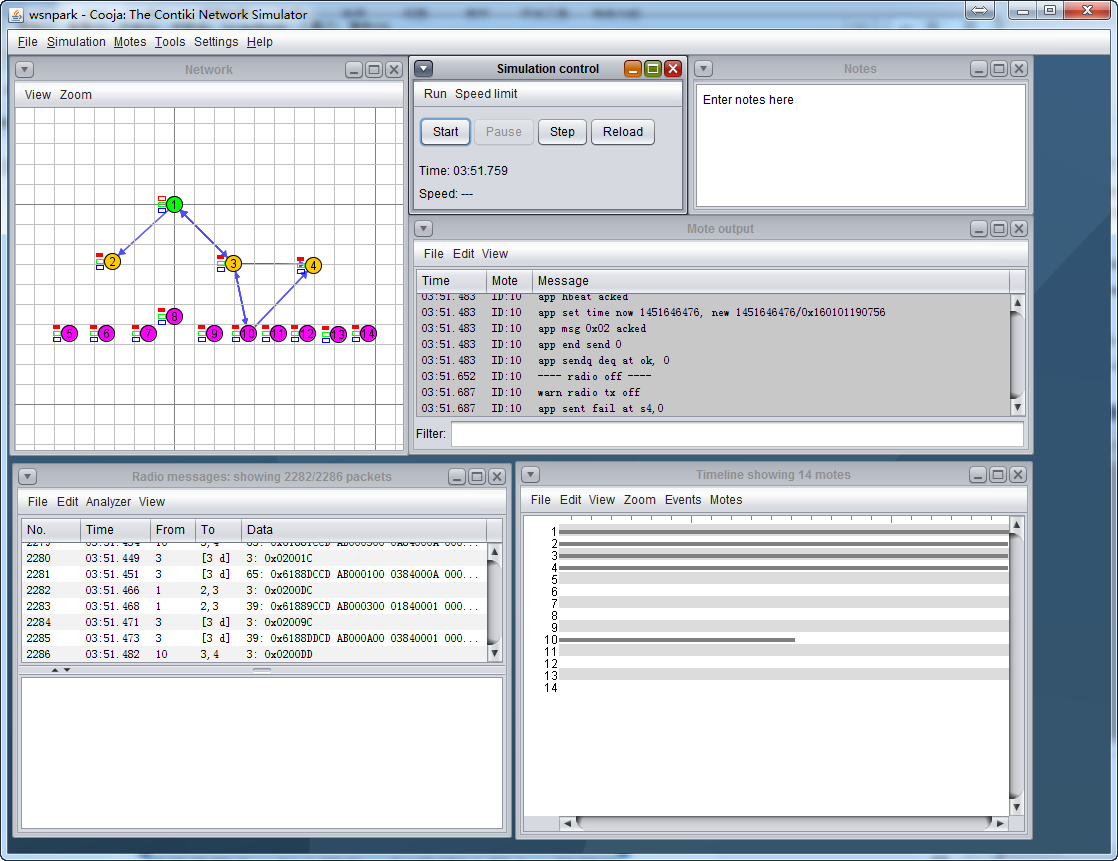


图 2 Cooja仿真示例

## Sniffer协议分析程序

Sniffer节点程序及配套GUI程序用于观察实际节点网络运行时交互的无线数据包，从而实现对无线网络的问题诊断。

Sniffer节点程序见apps/sniffer，需要在IAR EW430中进行编译并烧录到一个VD节点。

Sniffer GUI程序采用Java语言开发，需运行在安装Windows系统的笔记本电脑上。

Sniffer节点和GUI程序的详细使用方法参见文档《Sniffer使用手册》。

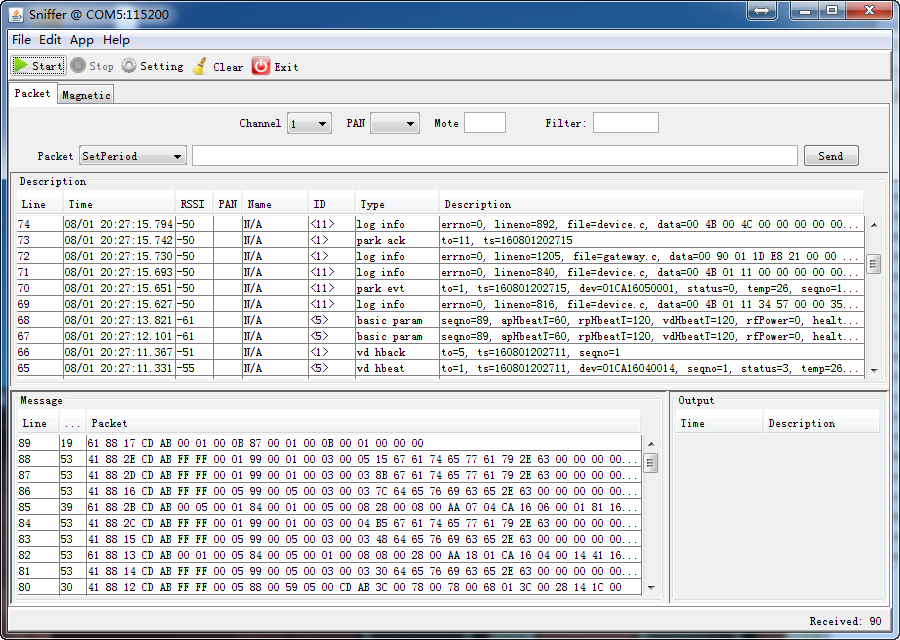


图 3 Sniffer GUI

要分析AP与服务器间的通讯还需使用GPRS串口监听程序，该程序同时以字符串和字节数组形式显示GPRS模块的串口输出，便于调试GPRS模块的工作状态；数据自动写入日志文件，可以查找诊断特定数据包是否正常收发。

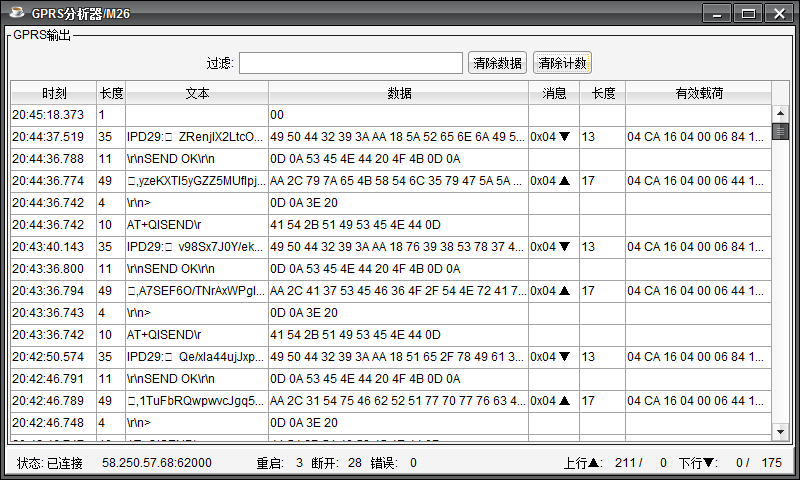


图 4 GPRS串口分析

## 演示后台系统程序

演示后台系统是一个桌面GUI程序，采用Java语言开发，可用于开发阶段接收小型节点网络的数据以测试系统功能和部分性能。该软件提供的功能包括：图形化展示网络部署和数据传输路径、列表展示泊位状态变化数据、曲线显示各VD节点的磁信号变化历史、列表显示网络上传和服务器下行数据用于调试、在界面中控制节点如重启、在界面中调整节点的运行参数等。该软件的详细用法参见文档《演示后台系统使用手册》。

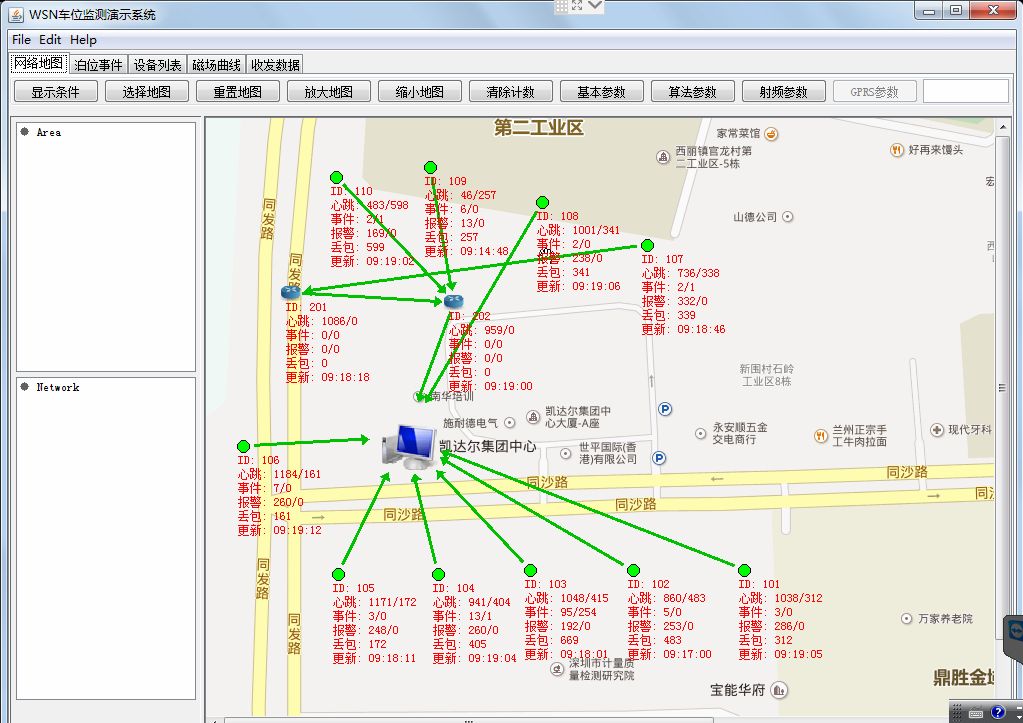


图 5 演示后台系统

## 完整开发环境搭建

要搭建本系统的完整开发环境，需按照以下几个步骤安装需要的开发软件：

1. 安装开发软件IAR for MSP430和烧录软件MSPFlasher。目前使用的版本是IAR for MSP430 6.30.2和MSPFlasher 1.3.11，两个版本的软件所要求的烧录器固件版本不同，MSPFlasher要求较新版本的固件，在MSPFlasher初次使用要求升级时应按照提示升级，之后在IAR中调试时会提示固件版本不匹配，无需升级也可以正常使用。
2. 安装Cygwin。Cygwin是Windows操作系统下运行的一个unix模拟环境，目前已有大量软件包移植到该环境中可运行。本开发环境需要使用其中的git、python、bash等软件自动处理编译过程的相关步骤，包括合并引导程序和主程序的输出msp430-txt文件、根据版本信息自动重命名输出文件、自动复制输出文件到发布或生产目录等，减少手动操作步骤、避免出错。可以从<http://www.cygwin.com>下载安装文件setup-x86.exe或setup-x86\_64.exe，配合附录的install.bat可以自动下载和安装本系统开发需要的软件包。
3. 安装Java SDK和Eclipse。如果需要修改本系统的相关上位机软件如Sniffer，需要安装Java开发SDK和集成开发环境如Eclipse。目前使用的版本是jdk-1.8.0-u60和eclipse-java-mars- R-win32。安装后应设置环境变量JAVA\_HOME=C:\Program Files (x86)\Java\jdk1.8.0\_60，同时设置PATH变量为PATH=C:\Program Files (x86)\Java\jdk1.8.0\_60\jre\bin;C:\Program Files (x86)\Java\jdk1.8.0\_60\bin;%PATH%，如果是32位系统，路径中无需“ (x86)”。
4. 设置相关环境变量。将cygwin路径和IAR路径添加到PATH变量，PATH=C:\cygwin\bin; C:\Program Files (x86)\IAR Systems\Embedded Workbench 7.2\common\bin;%PATH%。添加变量WVDS\_DIST指向发布目录，比如WVDS\_DIST=D:\WVDS\wvds-ota\dat\firmware；添加变量WVDS\_PROD指向生产目录，比如WVDS\_PROD=D:\WVDS\wvds-prog\Data。

环境变量的设置可以使用RapidEE软件，下载地址是[https://www.rapidee.com/en/ download](https://www.rapidee.com/en/download)，提供了更简单方便的界面进行设置。

## 程序编译和烧录

本系统的应用程序顶层代码在apps\wvds目录，各类型设备分别有对应的一个工程，VD设备程序对应device工程，RP设备程序对应router工程，AP设备程序对应gateway工程，引导程序对应boot工程，监听节点程序对应sniffer工程，更新节点程序对应updater工程。

每个工程有Debug和Release两种编译配置，使用Debug配置编译输出.d43格式文件，可以在IAR中联机调试，方便通过断点和单步调试诊断节点运行时的问题，但是这种方式下主程序没有和引导程序同时运行；使用Release配置时编译输出msp430-txt格式文件，通过使用prebuild.bat/sh在编译前自动更新头文件中的版本信息，通过使用postbuild.bat/sh自动在编译完成后合并引导程序和主程序的输出文件、根据版本信息重命名输出文件和复制到多个发布或生产目录。

使用Release配置编译时，链接阶段使用自定义的链接文件$PROJ\_DIR$\config\linker\ lnk430f5438a.xcl，编译结果将把代码和数据安排在自定义的地址区间（0x5C00-0xEFFF和0x10000-0x1FFFF），而不是默认的地址区间（0x5C00-0xFFFF和0x10000-0x1FFFF），编译输出文件必须与引导程序（位于地址区间0xF000-0xFFFF）同时使用才能够正常运行。在$PROJ\_DIR$\Release\Exe目录下已添加了对应的引导程序文件boot.txt，以VD设备为例，在apps\wvds\device\EW430\Release\Exe目录下有boot.txt，Release编译后生成apps\wvds\ device\EW430\Release\Exe\device.txt，通过自动执行postbuild.bat/sh，将把boot.txt和device.txt合并为device-boot.txt，该文件可以直接用烧录软件烧录到设备正常运行且支持无线更新。

为了进一步简化编译步骤，在apps\wvds目录下添加了build.bat脚本，双击运行后将出现如图 6的菜单，在其中选择要编译的工程对应的数字并按回车后就可以调用IAR命令行工具编译对应的工程，选择a并回车后则会编译本系统关联的所有工程。

按照Release配置编译后的输出文件要烧录到设备时操作步骤也较为繁琐，因此也添加了烧录脚本WVDS.bat简化操作。WVDS.bat和命令行烧录软件MSPFlasher的文件打包在同一目录下，假定为D:\WVDS\wvds-prog\，该目录下有子目录Data，当环境变量WVDS\_PROD指向D:\WVDS\wvds-prog\Data时，编译后postbuild.bat将自动复制文件如device-boot.txt到该目录。双击WVDS.bat后，在如图 7的菜单中选择欲烧录的文件对应的数字并按回车后将调用MSPFlasher自动烧录Data目录下对应的文件。

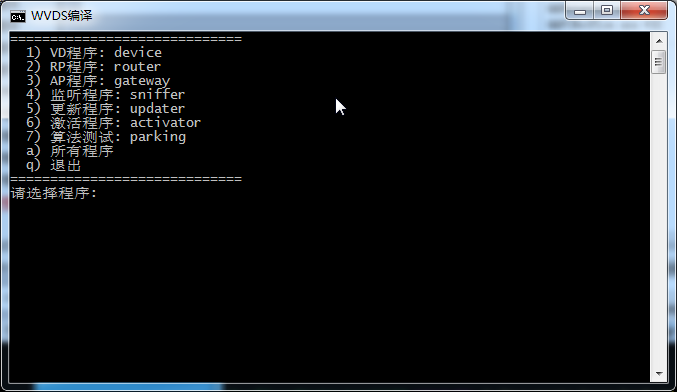


图 6 菜单式编译WVDS相关工程

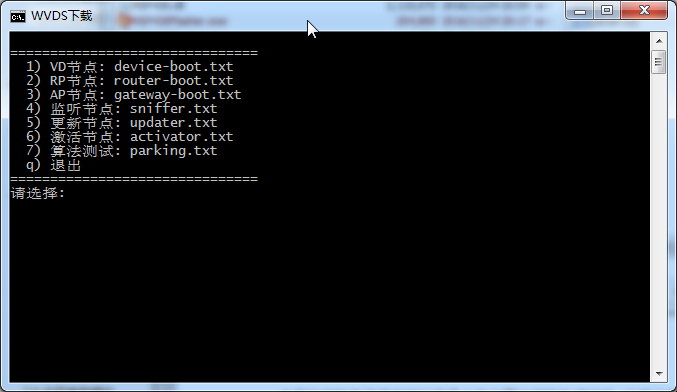


图 7 菜单式烧录WVDS相关程序

boot工程包含了各设备的引导程序，VD和RP、AP的引导程序仅于struct BOOT中部分域的取值不同，BOOT.role指明了设备类型，BOOT.flash指明了新固件存放的外部Flash的编号。在主文件main.c头部有如下代码段：

#define BOARD\_CADRE1120\_VD 1

#define BOARD\_CADRE1120\_RP 0

#define BOARD\_CADRE1120\_AP 0

编译时应根据适用的设备类型定义其中一个的值为1、其它为0，按照Release配置编译后将会调用postbuild.bat将输出文件boot.txt自动复制到apps\wvds\{device,router,gateway}\ EW430\Release\Exe目录下。后续再编译device等工程时就会用新版本的boot.txt与主程序输出文件如device.txt进行合并。

# 软件工程架构

本系统的软件工程包括以下几个协议层次的较多源代码，以下分节进行说明。以下描述中出现的子目录路径或文件路径均以源码根目录的相对路径表示。

## App层

本系统中定义了三种节点角色：AP（网关）、RP（中继器）和VD（车位检测器），它们分别对应不同的嵌入式主程序。

|  |  |  |  |
| --- | --- | --- | --- |
| 节点角色 | 源码目录 | 源码文件 | 说明 |
| VD | apps/wvds/device | apps/wvds/device/device.c | VD主线程文件 |
| apps/wvds/device/VehicleDetection.c | 车位检测算法文件 |
| RP | apps/wvds/router | apps/wvds/router/router.c | RP主线程文件 |
| AP | apps/wvds/gateway | apps/wvds/gateway/gateway.c | AP主线程文件 |
| apps/wvds/gateway/sim-server.c | 仿真时模拟服务器 |
|  | apps/wvds | apps/wvds/app.c | VD/RP/AP公用文件 |
| apps/wvds/cfs-flash.c | VD/RP/AP公用文件 |

表 1 App层文件列表

节点上电后，程序入口main()在节点平台对应目录下的\*-main.c文件中，本系统平台下为platform/mist-exp5438/contiki-exp5438-main.c。VD/RP/AP的主线程文件中都将该线程声明为AUTOSTART的线程，在main()函数中执行完必要的启动过程调用之后，将会启动声明为AUTOSTART的线程，即VD/RP/AP的主线程都被启动，随后该线程将打开需要使用的通信channel和相关的其他线程。

## NWK层

节点形成网络和支持多跳转发数据需要使用Contiki系统网络层（NWK）的较多相关组件，目前本系统仅使用称之为Rime栈的组件，不支持IP协议。这些组件之间具有较多的依赖，图 8表示了部分组件之间的依赖关系。

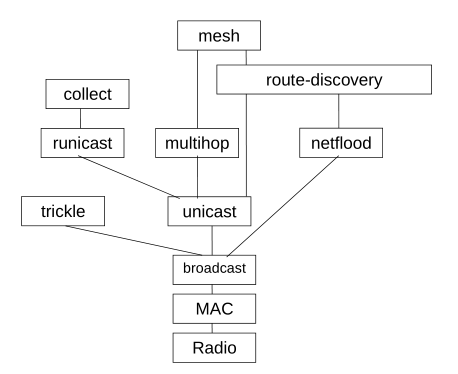


图 8 Rime协议栈部分组件依赖关系

NWK层需使用到的文件见contiki/core/net目录和contiki/core/net/rime目录，下表对主要用到的组件文件进行简要说明：

|  |  |
| --- | --- |
| 源码文件 | 功能说明 |
| contiki/core/net/rime/mesh.c | Mesh网络组件，依赖route-discovery发现到目标节点的路由，依赖multihop进行数据的多跳转发 |
| contiki/core/net/rime/multihop.c | 多跳转发组件，依据指定的网络路径将数据转发给下一跳节点 |
| contiki/core/net/rime/route-discovery.c | 路由发现组件，交换路由查询和响应消息实现源节点对目标节点传输路径的发现 |
| contiki/core/net/rime/route.c | 路由表维护组件，支持增加/替换/删除路由表项，支持存储路由表和重启后加载 |
| contiki/core/net/rime/trickle.c | 全网参数组件，用于维护全网需要保持一致的参数集合 |
| contiki/core/net/rime/unicast.c | 单播组件，向一个特定邻居节点发送数据包 |
| contiki/core/net/rime/broadcast.c | 广播组件，向所有邻居节点发送数据包 |
| contiki/core/net/rime/chameleon-bitopt.c | 数据包头部的生成和解析组件 |
| contiki/core/net/netcmd.c | 下行命令组件，用于网关向网内节点发送命令，支持休眠的VD节点 |
| contiki/core/net/neighbor.c | 邻居管理组件，记录邻居节点的地址和相关信息 |
| contiki/core/net/packetbuf.c | 数据包生成和解析组件 |
| contiki/core/net/queuebuf.c | 数据包缓存和再使用组件 |

表 2 网络层文件列表

## MAC层

|  |  |
| --- | --- |
| 源码文件 | 功能说明 |
| contiki/core/net/mac/csma.c | CSMA协议实现 |
| contiki/core/net/mac/nullrdc.c | RDC层实现，简单无休眠支持 |
| contiki/core/net/mac/framer-802154.c | 802.15.4数据帧生成和解析 |
| contiki/core/net/mac/frame802154.c | 802.15.4数据帧生成和解析 |

表 3 MAC层文件列表

## HAL层

|  |  |  |
| --- | --- | --- |
| 分类 | 源码文件 | 功能说明 |
| 射频 | dev/cc1120/cc11xx.c | CC1120驱动程序主文件 |
| dev/cc1120/cc1120-msp-arch.c | CC1120驱动程序硬件适配文件 |
| dev/cc1120/cc1120-config.h | CC1120驱动程序射频参数设置文件 |
| 磁传感器 | dev/mmc3316/mmc3316.c | MMC3316磁传感器驱动程序主文件 |
| dev/hmc5983/hmc5983.c | HMC5983磁传感器驱动程序主文件 |
| dev/qmc5883/qmc5883.c | QMC5883磁传感器驱动程序主文件 |
| GRPS | dev/m26/m26.c | M26 GPRS模块驱动程序主文件 |
| platform/cadre1120/m26-arch.c | M26 GPRS驱动程序硬件适配文件 |
| Flash  存储 | dev/m25pe/m25pe.c | M25PE Flash存储器驱动程序主文件 |
| dev/m25pe/c2195.c | M25PE官方驱动文件 |
| dev/m25pe/Serialize.c | M25PE官方驱动文件 |
| platform/cadre1120/m25pe-arch.c | M25PE驱动程序硬件适配文件 |
| RTC | dev/ds3231/ds3231.c | DS3231 RTC芯片驱动程序主文件 |
| platform/cadre1120/ds3231-arch.h | DS3231 RTC芯片驱动程序硬件适配文件 |
| 电池电压 | platform/cadre1120/battery-sensor.c | 电池电压检测 |
| 充电电压 | platform/cadre1120/solarbat-sensor.c | 太阳能电压检测 |
| 串口 | platform/cadre1120/dev/uart.c | 串口通信驱动封装文件 |
| platform/mist-exp5438/uart1x.c | 串口UART1的驱动程序 |
| platform/mist-exp5438/uart3x.c | 串口UART3的驱动程序 |

表 4 HAL层文件列表

# 通讯协议

## AP与服务器间通讯

AP与服务器之间的通讯要按照表 5定义的帧结构进行数据的封装，在后续描述中称为WVDS数据帧。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 帧头 | | 有效载荷（需加密和编码） | | | 帧尾 | |
| 起始字 | 数据长度 | 设备编码 | 命令字 | 消息内容 | 校验码 | 结束字 |
| 1 字节 | 1字节 | 6字节 | 1字节 | N字节(可变) | 2字节 | 1字节 |
| 固定字节数 2 | | 可变字节数7+N | | | 固定字节数3 | |
| 起始字 | 固定为0xAA | | | | | |
| 数据长度 | 有效载荷经加密和编码后的字节数，包括“设备编码”、“命令字”和“消息内容”。 | | | | | |
| 设备编码 | 具体定义见章节2.2。上行数据中填写该数据源节点的设备编码，例如VD节点的心跳数据0x02消息就填入该VD的设备编码；下行数据中填写该数据目标节点的设备编码，例如网关时钟同步0x05消息的响应中就填写目标AP的设备编码，检测参数设置0x26命令中就填写目标VD的设备编码。前端服务器处理接收到的消息时应该先将该设备编码转换为8字节的MAC地址再处理。 | | | | | |
| 命令字 | 具体定义见章节2.3。 | | | | | |
| 消息内容 | 具体定义见章节3。 | | | | | |
| 校验码 | 采用CRC-16校验方式，校验内容包括“数据长度”、“设备编码”、“命令字”、“消息内容”四个字段，根据多项式X^16+X^12+X^5+1计算，计算初始值为0x0000。传输时高字节在前。 | | | | | |
| 结束字 | 固定为0xFF | | | | | |

表 5 AP与服务器通讯的帧结构

### 消息列表

根据系统所需实现的功能，表 5定义的数据帧中的“消息内容”部分进一步定义以下列表 6中的多种消息。

|  |  |  |  |
| --- | --- | --- | --- |
| **操作内容** | **功能关键字** | **发起方** | **补充说明** |
| 车位检测器事件 | 0x01 | VD | VD在检测到泊位的有无车状态发生变化时向FES发送 |
| 车位检测器心跳 | 0x02 | VD | VD按一定周期向FES发送本数据 |
| 中继器心跳 | 0x03 | RP | RP按一定周期向FES发送本数据 |
| 网关心跳 | 0x04 | AP | AP按一定周期向FES发送本数据 |
| 网关时钟同步 | 0x05 | AP | AP主动向FES发送本请求以获得当前中国标准时间 |
| 保留 | 0x06-0x0F |  | 为深圳标准可能增加的命令保留 |

|  |  |  |  |
| --- | --- | --- | --- |
| **操作内容** | **功能关键字** | **发起方** | **补充说明** |
| AP入网认证 | 0x11 | AP | AP上电后通过GPRS连接向FES注册自身 |
| GPRS网络参数配置 | 0x14 | FES | FES向AP发送使之更改IP和PORT以连接新的FES |
| RF 参数查询或配置 | 0x15 | FES | FES向AP查询或设置其子网的射频频率、通道、发送功率 |
| 节点上线掉线报告 | 0x16 | AP | AP发现某RP/VD长时间无上报数据时报告其掉线，某RP/VD掉线后AP又收到其数据时报告其上线 |
| 子网锁定或解锁 | 0x17 | FES | FES向AP发送以锁定或解锁该子网，锁定时不允许新节点加入 |
| 软件更新数据 | 0x18 | FES | FES向AP发送本数据包逐块把新固件文件传输到AP上 |
| 子网软件更新执行 | 0x19 | FES | FES向AP发送启动AP向子网内更新已下载的新程序文件 |
| 保留 | 0x1A-0x1F |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **操作内容** | **功能关键字** | **发起方** | **补充说明** |
| VD/RP入网认证 | 0x21 | VD/RP | VD/RP入网时向AP报告自己的身份信息以获得认证资格 |
| 报警数据 | 0x22 | AP/RP/VD | AP/RP/VD按一定周期检查各硬件模块是否正常工作，发现故障时主动向FES报告故障码 |
| 历史泊位事件报告 | 0x23 | AP | AP恢复与FES间的通信后把中断通信期间存储在节点Flash中的历史泊位事件数据打包发送给FES |
| 磁场信号波动报告 | 0x24 | VD | VD在磁场信号发生一定程度的波动时向FES报告波动数据 |
| 基本参数查询或配置 | 0x25 | FES | FES向AP发送本命令查询或配置该子网的基本参数，如心跳周期、报警阈值、数据重发次数等 |
| 检测参数查询或配置 | 0x26 | FES | FES向VD发送本命令查询或配置泊位状态检测算法参数，如检测时间间隔、检测阈值等 |
| 保留 | 0x27-0x30 |  |  |
| 远程重启 | 0x31 | FES | FES向AP/RP/VD发送本命令使其重新启动 |
| 恢复出厂设置 | 0x32 | FES | FES向AP/RP/VD发送本命令使其恢复出厂设置 |
| VD设备激活 | 0x33 | VD | VD检测到强磁时向激活器发送以请求激活 |
| VD维护连接建立 | 0x34 | VD | 后续维护时VD检测到强磁时向激活器发送以开始通信 |
| VD维护连接断开 | 0x35 | ACT | 激活器完成维护操作后向指定VD发送以结束通信 |
| VD重新标定 | 0x36 | ACT | 激活器向指定VD发送使其重新标定背景磁场 |
| 保留 | 0x37-0x3F |  |  |

表 6 消息列表

更详细的消息内容描述参见文档《WVDS通讯协议》。

### 加密和编码

表 5定义的数据帧中“有效载荷”需要再进行AES加密和base64编码。根据通讯协议细则填充了消息内容之后，加密和编码按以下步骤进行：

1. 进行数据填充。计算有效载荷以字节计数的长度n，如果n不是16的倍数，m=16 - (n % 16)，即m是n离最近的16的倍数的差值，有效载荷的尾部需补充m个数值m；如果n正好是16的倍数，则需要在有效载荷的尾部补16个数值16；
2. 调用AES加密API对第1步填充后的数据块进行加密，加密采用128位密钥和CBC模式，CBC模式还要求128位的加密向量IV；
3. 调用base64编码API对第2步加密后的数据块进行编码。

在对有效载荷进行加密和编码后，整个数据帧的“数据长度”域应该设置为加密和编码后的有效载荷的长度，“校验码”也应该根据加密和编码后的有效载荷进行计算。

针对上述加密和编码后的消息进行解码和解密时，解密后得到的数据块需要再去除填充字节，方法是：查看解密后数据块的最后一个字节，假设其值为m，则去除数据块末尾的m个字节m。例如解密后最后一个字节是0x10，即16，则去除末尾的16个0x10。

## VD/RP与AP间通讯

为了简化系统的复杂度，VD/RP与AP间的通讯，其应用层数据也采用与AP和服务器间通讯协议相同的数据帧定义和消息内容定义。不同之处在于：1、不需要进行AES加密和Base64编码。应用层的明文数据帧在实际发送之前会由Rime协议栈的AES组件进行加密，并进一步封装为802.15.4的数据帧后在无线信道上发送；2、只用到AP与服务器间消息列表中的部分消息，而且这些消息是按照不同的发送方式到达VD/RP节点的。

VD/RP和AP间通讯所涉及的消息列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **操作内容** | **功能关键字** | **发起方** | **补充说明** |
| 车位检测器事件 | 0x01 | VD | VD在检测到泊位的有无车状态发生变化时向FES发送 |
| 车位检测器心跳 | 0x02 | VD | VD按一定周期向FES发送本数据 |
| 中继器心跳 | 0x03 | RP | RP按一定周期向FES发送本数据 |
| VD/RP入网认证 | 0x21 | VD/RP | VD/RP入网时向AP报告自己的身份信息以获得认证资格 |
| 报警数据 | 0x22 | VD/RP | VD/RP按一定周期检查各硬件模块是否正常工作，发现故障时主动向FES报告故障码 |
| 历史泊位事件报告 | 0x23 | AP | AP恢复与FES间的通信后把中断通信期间存储在节点Flash中的历史泊位事件数据打包发送给FES |
| 磁场信号波动报告 | 0x24 | VD | VD在磁场信号发生一定程度的波动时向FES报告波动数据 |

|  |  |  |  |
| --- | --- | --- | --- |
| **操作内容** | **功能关键字** | **发起方** | **补充说明** |
| 基本参数查询或配置 | 0x25 | FES | FES向AP发送本命令查询或配置该子网的基本参数，如心跳周期、报警阈值、数据重发次数等 |
| 检测参数查询或配置 | 0x26 | FES | FES向VD发送本命令查询或配置泊位状态检测算法参数，如检测时间间隔、检测阈值等 |
| 远程重启 | 0x31 | FES | FES向AP/RP/VD发送本命令使其重新启动 |
| 恢复出厂设置 | 0x32 | FES | FES向AP/RP/VD发送本命令使其恢复出厂设置 |

表 7 VD/RP和AP间通讯消息列表

### 消息封装

VD/RP和AP间通讯的应用层数据仍然采用如上定义的WVDS数据帧，但在具体发送时将会根据所使用的通信channel（类似于TCP/IP端口）进行不同的封装。由于网络层及以下已有CRC检查，子网间无线传输的数据省略了WVDS数据帧尾部的CRC和结束字符0xFF。

Contiki系统的Rime栈定义了通信channel，所有的上层数据都会经过Rime栈的不同组件进行发送，对应到不同的通信channel；channel的取值范围是1-65535，它也会包括发出的射频数据包中；接收节点将收到的数据包交给Rime栈底层，底层根据channel值相应调用上层的相关函数进行处理，经过多层调用最终可达用户的应用层处理函数。

不同channel上发送的消息根据其所用Rime组件的不同，对应用层数据增加了不同的网络层数据头。有的是通过attributes实现，比如multihop.c中multihop\_open()时将调用channel\_set\_attributes(channel, attributes)，其中attributes包括：

#define MULTIHOP\_ATTRIBUTES { PACKETBUF\_ADDR\_ESENDER, PACKETBUF\_ADDRSIZE }, \

{ PACKETBUF\_ADDR\_ERECEIVER, PACKETBUF\_ADDRSIZE }, \

{ PACKETBUF\_ATTR\_HOPS, PACKETBUF\_ATTR\_BIT \* 5 }, \

UNICAST\_ATTRIBUTES

因此multihop支持的多跳通信数据包在网络层数据头中增加了3个域：ESENDER、ERECEIVER和HOPS。有的则是通过模块内自定义消息，比如自实现的netcmd模块。

本系统定义了以下几个channel用于不同类型消息的交换：

|  |  |  |
| --- | --- | --- |
| 常量名 | Channel值 | 用途 |
| APP\_MESH\_CHANNEL | 0x84 | 实现网络中数据的多跳转发，是本系统VD/RP和AP间通信的主要channel。应用层载荷为WVDS数据帧。 |
|  | 0x85 | 实现路由信息的交换（由mesh自动附带创建）。 |
| APP\_NETCMD\_CHANNEL | 0x87 | 实现终端低功耗场景中下行命令的收发，定义了三种子类型消息实现VD请求并接收下行命令。应用层载荷为WVDS数据帧。 |
| APP\_TRICKLE\_CHANNEL | 0x88 | 实现全网参数的下发，通过不定时发送保证同样的数据集合被网络中所有节点接收到。应用层载荷为全网参数PIB结构体。 |
| APP\_UNICAST\_CHANNEL | 0x89 | 实现现场维护时与目标节点的点对点通讯。应用层载荷为WVDS数据帧。 |
| LOG\_CHANNEL | 0x99 | 实现发送运行时错误/警告/报告消息，供监听后诊断节点或网络运行时的问题。 |

表 8 系统使用channel列表

# OS架构与移植

## process调度器

Contiki中的代码可以运行在下列两种执行上下文之一：合作式或者抢占式。合作式代码按顺序运行，抢占式代码可以暂停正在运行的合作式代码。Contiki中的进程运行在合作式上下文中，而中断和实时定时器运行在抢占式上下文中。

所有的Contiki程序都被叫做进程(process)。一个进程是Contiki系统中被常规执行的一个代码片段。当系统启动时，或者一个包含进程的模块被加载到系统中时，进程开始运行。当一些事件发生时，进程会运行，比如一个定时器到期了，或者有一个外部事件产生。

Contiki的进程由两部分组成：进程控制块和进程线程。进程控制块存储在内存中，它包含进程运行时的信息，比如进程名、进程状态、指向进程线程的指针。进程线程是存储在ROM中的一个代码块。

进程线程包含进程的代码。进程线程是一个单一的protothread，由进程调度器调度。protothread提供了一种很好的方法，可以让C函数在没有传统线程内存负载的情况下，以类似线程的方式运行。一个进程线程的例子如下：

PROCESS\_THREAD(hello\_world\_process, ev, data)

{

PROCESS\_BEGIN();

printf("Hello, world\n");

PROCESS\_END();

}

protothread可以看做是一个常规C函数。该函数的使用两个特殊的宏作为开始和结束：PT\_BEGIN()和PT\_END()。在这两个宏之间，可以使用一系列的protothread函数。Contiki进程中使用的protothread语句与上述纯protothread语句有微小的差异，Contiki进程中使用的进程相关的protothread宏：

PROCESS\_BEGIN(); // Declares the beginning of a process' protothread.

PROCESS\_END(); // Declares the end of a process' protothread.

PROCESS\_EXIT(); // Exit the process.

PROCESS\_WAIT\_EVENT(); // Wait for any event.

PROCESS\_WAIT\_EVENT\_UNTIL(); // Wait for an event, but with a condition.

PROCESS\_YIELD(); // Wait for any event, equivalent to PROCESS\_WAIT\_EVENT().

PROCESS\_WAIT\_UNTIL(); // Wait for a given condition; may not yield the process.

PROCESS\_PAUSE(); // Temporarily yield the process.

contiki/core/sys/process.c实现了进程调度器，它的作用是调用进程。进程调度器通过调用实现进程线程的函数来调用进程。Contiki中的所有进程被设计为响应传递到进程中的事件，或者响应进程请求的轮询。进程调度器在调度进程的时候会将事件标识符和一个不透明指针传递到进程中。

在Contiki中，当进程接收到一个事件后就会运行。Contiki中有两种事件：异步事件和同步事件。当一个异步事件被发出时，通过函数process\_post()发出，该事件被放到内核中的事件队列中，并在一段时间后被传递到接收进程中。当一个同步事件被发出时，该事件被立即传递到接收进程中。轮询请求是一个特殊的事件。进程可以通过调用函数process\_poll()请求被轮询。

以下调用关系简单说明了主循环中如何驱动进程调度器以及process.c如何调用call\_process()到执行某个进程内的相关代码：

main() # platform/mist-exp5438/contiki-exp5438-main.c

while(1)

process\_run() # contiki/core/sys/process.c

do\_poll() # contiki/core/sys/process.c

call\_process(p, PROCESS\_EVENT\_POLL, NULL)

do\_event() # contiki/core/sys/process.c

do\_poll()

call\_process(p, ev, data) # contiki/core/sys/process.c

p->thread(&p->pt, ev, data)

## 时钟及timer

嵌入式程序必然要定时执行一些任务，在Contiki系统中，这是通过本地时间结合etimer和ctimer实现的。

Contiki系统定义了一个常数CLOCK\_SECOND，节点的本地时间不是以秒为单位，而是以系统节拍为单位，CLOCK\_SECOND个节拍的时间等于1秒。例如CLOCK\_SECOND默认为128，以一个32位整数count表示节点本地时间，则count增加了128表示标准时间经过了1秒。

根据CLOCK\_SECOND常数，节点初始化时clock.c文件必须要正确初始化某个硬件定时器，使之每1/CLOCK\_SECOND秒产生一次中断（后续称之为时基中断），在中断处理函数中，系统要将本地时间加1，并且检查是否有定时任务已经到时需要执行。如果修改CLOCK\_SECOND的值，必须相应修改时基中断的间隔。

etimer是Contiki系统用来执行定时任务的一种软件定时器。etimer需要在某个线程的上下文内使用，通过etimer\_set()等API可以设置一个定时时间，系统将在经过该时间后调用该线程使之运行定时后想完成的操作。etimer.c中实现了线程etimer\_process，它提供etimer\_set()等API，负责维护系统运行时的多个etimer、在每个etimer计时结束时调用相应的线程。在每次时基中断时，需要检查是否有etimer已计时结束，如果是，需要调用etimer\_process进行对应的处理。

ctimer是Contiki系统执行定时任务的另一种方式。ctimer可以不在线程的上下文中调用，调用其相关的API时需要指定一个回调函数的指针作为参数，在计时结束时，该函数将被执行。ctimer依赖于etimer进行相关的实现。

## 通信协议栈

Contiki系统的通信协议栈定义了几个重要的宏指向各层的驱动程序结构体，它们分别是：NETSTACK\_NETWORK、NETSTACK\_MAC、NETSTACK\_RDC、NETSTACK\_RADIO。这些宏都指向某一个结构体，该结构体的成员均为函数指针，定义了该模块作为通信协议栈的网络层或MAC层或物理层需要提供的几个关键API。

增加一个新的节点平台时都需要在该平台的contiki-conf.h文件或包含的其他头文件中指定上述几个宏默认的对象，基于该平台开发的应用程序也可以在自己的项目头文件中改变contiki-conf.h中的默认设置。通过改变以上这些宏指定的对象，就可以改变所使用的协议栈相应层的具体实现。

目前本系统采用的协议栈各层实现从上到下如下表所示：

|  |  |
| --- | --- |
| NETSTACK\_NETWORK | rime\_driver |
| NETSTACK\_MAC | csma\_driver |
| NETSTACK\_RDC | nullrdc\_driver |
| NETSTACK\_RADIO | cc11xx\_driver |

NETSTACK\_NETWORK指定网络层的具体实现，目前支持的主要有：1) rime\_driver，Rime网络层，可支持mesh网络；2) sicslowpan\_driver，支持6LoWPAN网络；3) uip\_driver，支持IPv4网络；4) slipnet\_driver，支持通过串口连接运行SLIP接入IPv4网络。本系统采用rime\_driver。

NETSTACK\_MAC指定MAC层的具体实现，目前支持的主要有：1) nullmac\_driver，无MAC层的特殊控制；2) csma\_driver，采用CSMA机制。本系统采用csma\_driver。

NETSTACK\_RDC指定RDC层的具体实现。RDC层是Contiki特别定义的一层，介于MAC层和PHY层之间，其作用是统一控制射频的开启和关闭实现节点低功耗且满足MAC层通信需求。目前支持的主要有：1) nullrdc\_driver，无特别控制；2) contikimac\_driver，Contiki系统作者发表的协议，调试较复杂；3) xmac\_driver，基于X-MAC的简单节能协议；4) lpp\_driver，接收节点主动发送探测数据包引导发送节点发送。目前本系统采用nullrdc\_driver。

NETSTACK\_RADIO指定PHY层的具体实现，即采用哪种射频对应的驱动程序，必须与所使用的硬件节点上射频芯片一致。目前本系统采用cc11xx\_driver。

此外，无线数据的传输支持AES-128加密，目前是在RDC层实现的。

## 新节点平台移植

如果需要将Contiki系统移植到某种新的硬件节点，可参考以下步骤进行移植：

1. 查看contiki/cpu/目录下是否已经有对应的MCU支持，有支持尽量直接使用，需要时做必要修改，否则需要在contiki/cpu目录下增加子目录支持所使用的MCU；
2. 在contiki/platform/目录下增加一个子目录如foo存放该节点对应的平台基本文件，至少要增加文件contiki-conf.h、contiki-foo-main.c等文件；
3. 参考其他节点平台的主函数文件，修改新增节点的主函数文件contiki-foo-main.c，一般来说，需要增加节点的时钟、LED、射频等硬件的初始化，初始化process调度器和基本的系统线程如etimer\_process，再启动标记为自启动的线程，然后进入主循环通过process\_run()不断处理可能的事件；
4. 在时钟初始化时需要注意根据自定义的CLOCK\_SECOND设置正确的时基中断间隔，修改后先使用一个测试程序先进行必要测试。

## 射频驱动移植

如果改变了硬件节点使用的射频模块，需要更改射频驱动程序，可参考以下步骤：

1. 在dev目录建立一个子目录存放新射频驱动需添加的相关源代码文件；
2. 如果该射频芯片提供了官方驱动源码，可以将它们导入第1步建立的目录中，再通过编写射频主线程文件对其进行封装实现与contiki通信协议栈上层匹配的接口，如果没有相关代码则只能根据后续步骤从头开发；
3. 根据contiki/core/dev/radio.h中的API定义生成一个射频主线程文件，一般以射频芯片型号命名，以射频芯片cc1120为例，添加必须的空函数并定义如struct radio\_driver cc1120\_driver，在contiki-conf.h中将NETSTACK\_RADIO修改为cc1120\_driver；
4. 在第3步包括空函数的射频主线程文件基础上添加各API的具体实现，有相关代码时可以直接调用它们提供的API。要正确理解radio.h中定义的这些API向外提供的功能，不清楚时可参考其他射频驱动的实现，也要分析上层组件对这些API的调用，避免可能造成一些射频处理时序上的错误。

# 底层驱动流程图

## cc1120射频模块驱动

cc1120射频模块的驱动程序相关实现文件在目录dev/cc1120中。射频驱动的主要控制流程实现为一个单独的线程cc11xx\_process，在文件dev/cc1120/cc11xx.c中；其与硬件节点相关的部分在文件dev/cc1120/cc1120-msp-arch.c中实现，如果移植到其他不同硬件连接或不同MCU的节点上，只需要修改该文件提供等价功能即可；射频模块的大部分配置对应的寄存器值在文件dev/cc1120/cc1120-config.h文件中指定，射频工作频率、发送功率等配置对应的寄存器值则可能通过代码在运行时改变。

在实现射频驱动程序时，需要根据dev/radio.h提供一个struct radio\_driver的变量，该结构体变量的域均为驱动模块内的相关子函数的指针。cc11xx.c提供的驱动结构体如下代码段所示，平台的contiki-conf.h文件中将定义NETSTACK\_RADIO为cc11xx\_driver，相关调用代码中将使用NETSTACK\_RADIO.init()的形式进行调用：

const struct radio\_driver cc11xx\_driver = {

init,

prepare,

transmit,

send\_packet,

read\_packet,

channel\_clear,

receiving\_packet,

pending\_packet,

on,

off,

};

其中init()应完成射频的初始化，prepare()、transmit()、send\_packet()、channel\_clear()与发送数据有关，read\_packet()、receiving\_packet()、pending\_packet()与接收数据有关，on()、off()控制射频模块的开启/关闭以降低射频部分功耗。

cc1120射频驱动程序中关键的流程有三个：

1. 射频初始化，相关子函数的调用关系如下：

NETSTACK\_RADIO.init() # dev/cc1120/cc11xx.c: init()

cc1120\_arch\_init() # dev/cc1120/cc1120-msp-arch.c，初始化相关IO口和SPI接口

spi\_init() # platform/mist-exp5438/spix.c

reset() # dev/cc1120/cc11xx.c，重启射频并写入所有配置的寄存器值

strobe(CC11xx\_SRES) # 发送SRES指令重启射频

single\_write(CC11xx\_IOCFG3,…) # 写入寄存器IOCFG3的配置值

on() # dev/cc1120/cc11xx.c，开启射频

process\_start(&cc11xx\_process,…) # 启动cc11xx\_process

off() # dev/cc1120/cc11xx.c，关闭射频

cc1120\_arch\_interrupt\_enable() # dev/cc1120/cc1120-msp-arch.c，使能射频中断

calibrate\_manual() # dev/cc1120/cc11xx.c，进行射频校准

1. 射频发送，相关子函数的调用如下：

NETSTACK\_RADIO.send() # dev/cc1120/cc11xx.c: send\_packet()

prepare() # dev/cc1120/cc11xx.c，准备发送、缓存待发送数据

memcpy(packet\_tx,…)

transmit() # dev/cc1120/cc11xx.c，进行实际发送

strobe(CC11xx\_SIDLE) # 使射频进入IDLE状态

write\_txfifo(packet\_tx,…) # 将待发送数据写入TXFIFO

burst\_write(CC11xx\_TXFIFO,…) # 写入初次发送数据

strobe(CC11xx\_STX) # 发送STX指令开始发送

burst\_write(CC11xx\_TXFIFO,…) # 写入后续发送数据

BUSYWAIT\_UNTIL((state()==CC11xx\_STATE\_TX),…) # 等待发送进行

BUSYWAIT\_UNTIL((state()==CC11xx\_STATE\_TX),…) # 等待发送完成

值得注意的是，发送数据时不采用中断方式，即不是发出发送指令后在发送完成中断的处理函数中进行发送完成后的相关操作，而是等待并轮询cc1120的状态机寄存器直到发送完成。两种实现方式的不同将会显著改变相关的流程、极可能导致错误，不可随意改变。

1. 射频接收，相关子函数的主要调用如下：

ISR(PORT1, cc1120\_port1\_interrupt) # dev/cc1120/cc1120-msp-arch.c，射频中断处理函数

cc11xx\_rx\_interrupt() # dev/cc1120/cc11xx.c

read\_rxbytes() # 读取当前RXFIFO的字节数

burst\_read(CC11xx\_RXFIFO,…) # 读出首字节，为数据长度

input\_byte(byte)

burst\_read(CC11xx\_RXFIFO,…) # 读出后续数据字节

input\_byte(byte) # 交给接收protothread处理

PT\_BEGIN(&rxstate.pt)

rxstate.buf[rxstate.ptr++]=byte # 接收PT缓冲区输入字节

PT\_YIELD(&rxstate.pt)

flushrx() # CRC出错时清除RXFIFO并再进入接收模式

pending\_packet() # 有已接收数据待处理

frame802154\_parse(rxstate.buf,…) # 解析为802.15.4帧

send\_ack() # 需要时发送ACK

memcpy(packet\_rx,…) # 复制到接收缓冲区

process\_poll(&cc11xx\_process) # poll驱动线程稍后处理

pollhandler() # 驱动线程POLL事件处理函数

packetbuf\_clear() # 重置packetbuf

read\_packet() # 读取接收数据放入packetbuf

packetbuf\_set\_datalen() # 设置packetbuf中数据长度

NETSTACK\_RDC.input() # 交给RDC层模块处理

LPM4\_EXIT # 从LPMx模式退出

以上只列出了正常接收情况下的一些函数调用，完整的流程包括较多异常处理调用，需对源代码进行分析了解整个流程。

cc1120驱动程序中需要注意的一点是，发送数据包时，需要进行较长时间的延时才能使节点发射出大能量，测量节点功耗时可从电流表观察到大电流（最大可达约300mA）。发送过程对应的伪代码如下：

strobe(CC11xx\_STX); // 使射频进入发送模式

delay\_ms(100); // 延时100ms

write\_txfifo(packet, len); // 将待发送数据写入射频TXFIFO

strobe(CC11xx\_STX); // 使射频开始发送数据

该过程需注意两点：1、延时需要在使射频进入发送模式后再进行，即前两行代码的顺序不可调换；2、延时的长短与发送电流存在一定的正比关系，小于50ms时变化率较大，在50ms到100ms间变化率较小，大于100ms则电流无明显增长。

上述的发送延时导致两个问题：1、发送ACK时也需要进行延时，否则ACK接收节点如VD可能因信道不对称而无法接收到；2、在MAC/RDC层等待如ACK时需增加相应的时间，否则可能因时间短而无法接收到。

在平台配置文件platform/mist-exp5438/contiki-conf.h中新增了一个常量PACKET\_TIME，表示发送一个数据包所需的时间，其值应为最大长度的数据包在平台所采用空中波特率下完成发送所需的时间。在有上述延时的情况下，该时间还需要加上延时时间。PACKET\_TIME的值应该以系统节拍值表示，比如125ms对应为CLOCK\_SECOND/8。在代码中涉及到射频相关的时间时，应以PACKET\_TIME为基础来计算，则在系统更改空中波特率等情况下直接修改该常量的值就可以完成必要的更改。

## mmc3316磁传感器驱动

本部分仅用于第一阶段验证版节点，现已弃用，改用国产qmc5883磁传感器驱动。

mmc3316磁传感器的驱动程序的相关实现文件在目录dev/mmc3316中。

mmc3316磁传感器通过I2C接口与MCU进行通信，在本系统的VD节点上，2个磁传感器分别连接到P5.4/UCB1SCL、P3.7/UCB1SDA和P10.2/UCB3SCL、P10.1/UCB3SDA上，可以采用硬件I2C接口实现，但目前采用的实现方式是作为普通IO通过软件模拟I2C接口与mmc3316通信，更容易调整时序与mmc3316的要求相符。

由于VD节点上有2个传感器，本驱动提供的API均要求提供一个devid参数指明要针对哪一个传感器进行操作。在具体实现中，调用这些API时devid将被赋给驱动模块内的一个全局变量cur\_dev，模块内部的一些函数调用与该变量无关，直到最底层SCL\_1()/SCL\_0()、SDA\_1()/SDA\_0()这些子函数需要设置具体的IO口电平时，这几个子函数才根据cur\_dev设置当前使用传感器对应的IO口。

I2C接口的软件模拟和mmc3316磁传感器的接口时序可参见源代码和传感器的数据表。

## qmc5883磁传感器驱动

qmc5883磁传感器的驱动程序的相关实现文件在目录dev/qmc5883中。

qmc5883传感器也采用I2C接口与MCU进行通信，最新的1.1版本VD设计中连接到P10.1/UCB3SDA和P10.2/UCB3SCL管脚上，另外传感器有一个输出管脚指示是否采样数据已经准备好，它连接到P2.4管脚上可通过中断进行处理。

qmc5883传感器驱动采用硬件I2C接口的实现代码，即直接操作该接口的对应MCU寄存器，该接口为多协议接口，所以是通过设置作为I2C接口使用。

目前qmc5883传感器的驱动采用的是单次采样的方式，由检测算法决定何时需进行采样。每次采样完成后，应该使传感器进入standby状态以减少功耗。qmc5883有两个采样值范围，分别是±8G和±2G，可以通过定义QMC5883\_RANGE等于8或2分别采用对应范围；采样结果自动转换为以mGauss为单位。

启动采样后，传感器需耗时几个毫秒才能完成。为了实现更低的功耗，MCU在发送指令让传感器开始采样后，就结束任务，进入到休眠状态。之后有两种方式在传感器数据准备好时进行处理，一是在中断管脚有信号时经中断处理函数调用，二是使用ctimer在采样过程结束后调用，两种方式采用共同的子函数读取数据、转换单位并调用回调函数。经测试，目前采用第二种方式。分析电流采样结果可发现，每次采样对应两个峰值，中间有几毫秒时间节点返回了休眠状态。

## m26 GPRS模块驱动

m26 GPRS模块的驱动程序由dev/m26目录的驱动主文件m26.[ch]和platform/cadre1120目录下的硬件适配文件m26-arch.[ch]组成。

m26-arch.c文件提供了几个m26\_arch\_xxx()的基本API支持开启或关闭GPRS模块、向模块串口发送数据及注册串口收到数据的回调函数等。如果需要将m26的驱动移植到别的节点平台下使用，基本只需要修改m26\_arch.c中的相关API的实现即可完成。

m26.c是驱动程序的主文件，实现为一个独立的process。该驱动程序的实现要点包括：

1. 使用m26\_byte\_rcvd()作为串口中断的处理函数，其功能是将接收的字节放入模块内的字节环形队列供稍后处理。由于波特率为115200、速率较快，为了避免处理其他任务导致出现字节丢失，目前采用接收到第一个字节后，禁用串口接收中断和总中断，通过轮询方式接收当前数据帧直到结束，然后再使能串口接收中断和总中断，并poll驱动线程稍后处理；
2. 将接收字节环形队列中未处理的部分逐个字节输入m26\_handle\_byte()，该子函数使用状态机来处理当前的输入，每输入一个字节，根据当前状态和输入字节切换到下一个状态或保持当前状态，同时将该字节放入另一个数据帧字节数组；当状态切换到某种合法帧最后一个字节对应的状态值时，数据帧字节数组中即为待处理的合法数据帧，交由m26\_frame\_rcvd()进行处理。
3. 子函数m26\_frame\_rcvd()负责处理接收到的数据帧，它首先根据数据帧的内容分为多个分支进行处理，在每个分支内，又需要根据模块当前建立GPRS连接的中间状态进行不同的处理（典型的是GPRS模块返回的OK在不同状态下需要进行不同处理）；
4. 要驱动m26模块建立GPRS连接需要发送一系列指令，有时需要等待较长时间发送下一条指令，有时控制不成功需要重复发送指令，因此也采用状态机和ctimer实现对m26模块的控制。在每个建立连接中的中间状态下，首先启动ctimer以小间隔立即发送对应的控制指令，然后以该ctimer设置超时，如果超时前接收正常的m26模块响应数据，则该指令已成功，停止超时，转入下一状态，因此发送下一条指令；否则在ctimer到时后将重发当前指令。重复该过程直到已经成功能建立与服务器的GPRS连接。
5. M26的驱动程序的一个简化版本的状态图如图 9：

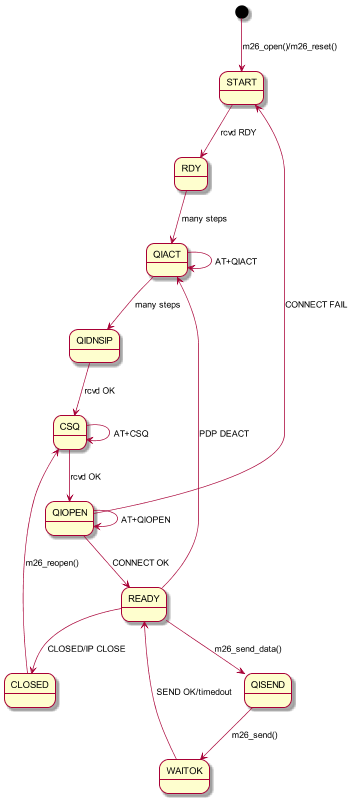


图 9 M26 GPRS模块驱动程序状态图

## m25pe存储器驱动

本系统使用m25pe系列Flash作为外部存储器，对应的驱动程序实现文件包括dev/ m25pe目录下所有文件和硬件适配文件platform/cadre1120/m25pe-arch.[ch]。

platform/cadre1120/m25pe-arch.[ch]定义了与硬件PCB匹配的相关宏，以设置对应的IO端口和操作SPI接口，移植到其他节点平台时基本只需要提供这两个文件的等价API。

m25pe驱动的实现使用了芯片厂商提供的官方代码，即dev/m25pe/c2195.[ch]和dev/m25pe/Serialize.[ch]，并添加了m25pe.[ch]进行封装对外提供API。

与mmc3316传感器的驱动程序类似，本驱动API包括参数devid指明使用哪个Flash，调用时被赋值给模块内的全局变量cur\_dev，在最底层的子函数中根据该值操作对应的SPI接口和IO端口。

为了满足低功耗需求，最新版本VD节点增加了对Flash存储器的电源控制，连接到MCU的P10.0管脚。在结束对Flash的访问后，首先应向Flash发送“深度睡眠”指令，然后拉低P10.0管脚断开给Flash的电源；在需要重新访问Flash时，拉高P10.0管脚给Flash重新上电，等待稍长时间如~8ms后，先发送“退出深度睡眠”指令给Flash，再进行所需的读写操作。m25pe驱动中低功耗相关的代码由预编译符号M25PE\_LPM包围，默认定义为0，可以定义为1来使能驱动的低功耗支持。

## ds3231 RTC驱动

本系统使用DS3231 RTC芯片作为实时时间模块，对应的的驱动程序实现文件包括dev/ds3231/ds3231.[ch]和硬件适配文件platform/cadre1120/ds3231-arch.h。

DS3231也采用I2C接口与MCU通信，与磁传感器驱动类似，采用软件模拟方式实现，细节可查看源码进行分析。

## ADC驱动

本系统使用ADC来检测电池电压和太阳能电池提供的充电电压。具体的检测方式为将待检电压通过适当的电阻分压输入到MCU的某个ADC端口，通过对应的MCU内部ADC采样后转换为对应的电压值。

电池电压和太阳能电池电压分别抽象为软件传感器组件platform/cadre1120/dev/ battery-sensor.c和platform/cadre1120/dev/solarbat-sensor.c。按照Contiki系统传感器组件的一般实现方法，在configure()中设置相关IO口和ADC通道等参数，在value()中启动ADC进行采样，并转换为最终的电压值。为提高准确性，采用多次采样、去除最大最小值后取平均值进行电压值的计算，计算中还必须乘上正确的分压比例值。

# 主函数流程图

## 数据消息抽象

根据前述定义的AP与服务器间通讯协议和RP/VD与AP间通讯协议，定义了以下的几个结构体来表示数据消息：

struct app\_msg\_header {

uint8\_t beg;

uint8\_t len;

uint8\_t mac[6];

uint8\_t cmdop:6;

uint8\_t rep:1;

uint8\_t dir:1;

};

struct app\_msg\_footer {

uint8\_t crc[2];

uint8\_t end;

};

struct app\_msg {

struct app\_msg\_header header;

uint8\_t data[APP\_DATA\_MAXLEN];

struct app\_msg\_footer footer;

};

其中struct app\_msg\_header表示表 5数据帧定义中不定长的消息内容之前的部分，struct app\_msg\_footer表示消息内容之后的部分，struct app\_msg表示完整的消息。

struct app\_msg结构体中的data部分根据不同的消息内容进一步可定义为对应的结构体，例如车位检测器事件消息对应如下定义：

struct app\_data\_parkevt {

uint8\_t tstamp[TSTAMP\_LEN];

uint8\_t status;

int8\_t temperature;

uint8\_t seqno;

int16\_t magneticX;

int16\_t magneticY;

int16\_t magneticZ;

uint8\_t batVoltage;

uint8\_t batQuantity;

};

## VD节点主程序

apps/wvds/device/device.c为VD节点的主线程文件，它的主要功能包括启动相关的子线程、建立与AP间的无线连接、调用mesh组件多跳发送数据、周期性采样磁场信号并交给算法组件处理、启动多个软件定时器执行心跳等任务、接收来自AP的消息并进行相应处理、关键数据存储及再发送。VD主程序的主要流程可见图 10。

VD主程序有以下几个要点需要注意：

### 磁场采样与算法处理

VD节点需周期性采样磁场信号并交给检测算法模块进行处理。VD主线程使用一个专用etimer进行周期性采样，当该定时器到时，调用磁传感器驱动程序的API MMC3316\_Read()进行采样，采样完成后，数据保存在驱动程序模块的MMC3316[]全局变量中，再调用检测算法模块的API Parking\_Algorithm()进行判决。

检测算法模块根据当前的磁场信号是否处于平稳状态会自适应调整采样的周期，通过主线程中的Set\_AMR\_Period()进行设置。需注意的是，检测算法模块中的时间以毫秒为单位，而其他模块以Contiki的ticks为单位，因此在Set\_AMR\_Period()中需要根据CLOCK\_SECOND进行时间的转换。

VD主线程中使用变量park\_s保存检测算法的输出状态。每次采样完经算法判决后，输出状态由临时变量status保存，与park\_s比较，如果两者不同，则生成泊位事件消息向AP发送，之后将status赋给park\_s。

### 关键数据消息存储

泊位在有车辆进出时，VD节点采样磁场信号并经算法模块检测到有车出入后会生成泊位事件数据，为保证这样数据的可靠性，将会先保存到外部Flash存储器之后再发送给AP。涉及到的API包括：

int app\_save\_msg(uint8\_t cmdop, uint8\_t \*data, uint8\_t len);

int app\_load\_msg(uint8\_t \*cmdop, uint8\_t \*buf, uint8\_t \*len);

int app\_erase\_msg(uint16\_t idx);

int app\_send\_saved\_msg(void);

在保存和加载消息时，还需要保存一些辅助信息，因此定义了如下结构体：

struct msgmeta {

uint8\_t id; // 消息的命令字

uint8\_t len; // 消息的字节长度

uint16\_t crc; // 消息的CRC-16校验和

uint32\_t ts; // 消息的时间戳，unix时间

uint8\_t erased; // 消息是否已被擦除。0:未擦除，非0:已擦除

uint8\_t reserved[7];

};

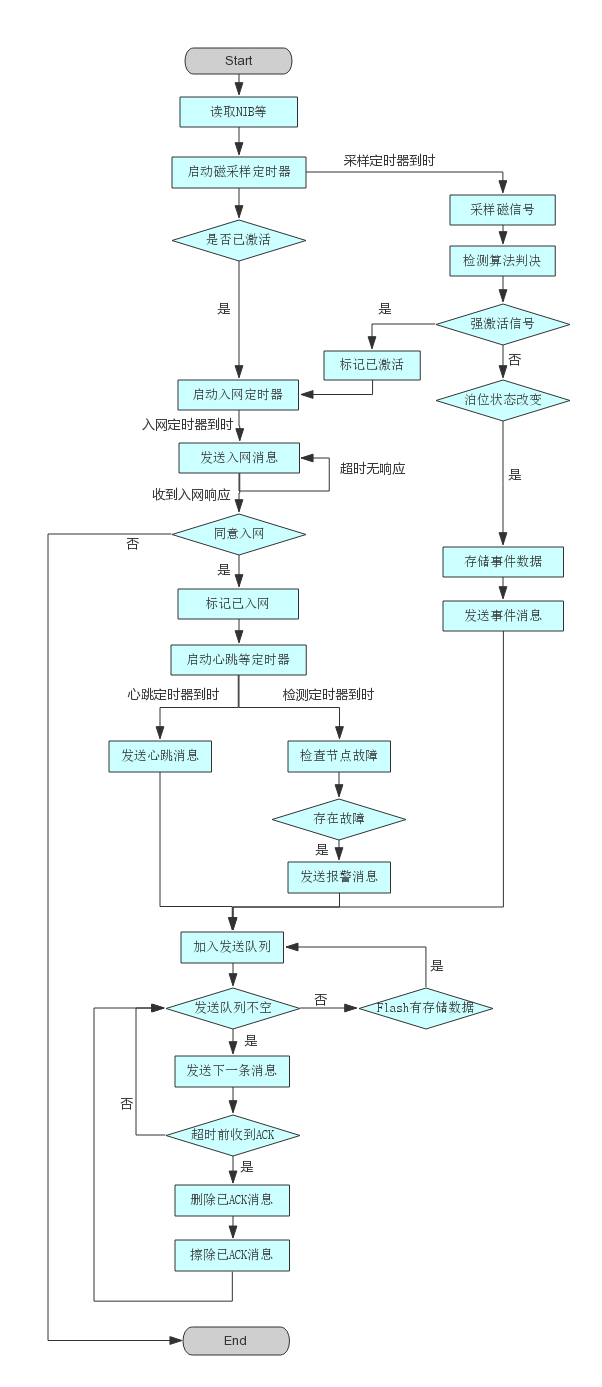


图 10 VD主线程流程

其中，crc域用于在加载消息时检查是否存在数据错误；ts域用于记录保存消息时的节点本地时间戳，在节点的本地时间因时间同步进行调整后需要进行更改以修正消息数据中的时间戳；erased标记用于表示消息是否已被擦除，在擦除消息时并未真的擦除保存的消息数据，而是只改写了erased标记。

在消息存储时，为每个消息保留128字节的Flash空间，其中前16字节用于保存对应的struct msgmeta，后续112字节用于保存实际的消息数据。

### 外部Flash作为环形队列使用

为了实现外部Flash存储器写入寿命的最大化利用，存储时将分配给关键数据消息的区间作为环形队列来使用。相关的细节如下所述：

定义了结构体MSGS用来记录Flash环形队列用于存储关键消息数据时的使用信息。

struct MSGS {

uint16\_t magic; // 初始化标记

uint16\_t count; // 已存储消息个数

uint16\_t readptr; // 读取索引

uint16\_t writeptr; // 写入索引

uint8\_t reserved[8];

uint8\_t messages[0];

};

用于存储关键消息数据的Flash空间定义为从地址NV\_MSGS\_ADDR开始，大小假设为20kB，则该段空间按下表所示进行分配，第一列是相对NV\_MSGS\_ADDR的相对地址，第二列是存储的数据内容。起始地址0x0000处存储struct MSGS msgs，保存消息Flash队列的状态信息；从地址0x0080起为保存的消息数据，每条消息的预留空间是128字节。

|  |  |
| --- | --- |
| 0x0000 | struct MSGS |
| 0x0080 | Msg[1] |
| 0x0100 | Msg[2] |
| …… | …… |
| 0x5000 | Msg[159] |

根据以上方案，有新消息需要存储时，根据struct MSGS的writeptr可计算出新消息的写入地址，即：addr = NV\_MSGS\_ADDR + 0x80 + (msgs.writeptr \* 128)，新消息的辅助数据和数据内容就写入addr开始的128字节空间内，同时把msgs.writeptr和msgs.count增加1；当某条消息已经成功完成发送需要删除时，根据该消息在Flash空间的索引值，可计算对应地址addr = NV\_MSGS\_ADDR + 0x80 + (index \* 128)，按照地址值addr即可更改辅助数据中的erased标记表明消息已擦除，如果index等于msgs.readptr，把msgs.readptr加1，msgs.count减1。按照以上的操作方式，关键消息数据将在该段Flash空间内均匀地写入。

### 支持优先级的消息队列

为了解决有时产生了新数据包要发送而网络层尚未完成上一个数据包发送的问题，VD程序中采用消息队列来存储要发送的数据包。在进行消息发送时，首先发送队列头的消息，在该消息发送成功或者失败达到最大重传次数后，才删除队列头的消息，并在队列不为空时发送队列新的头消息。

同时，在多个消息存入队列要发送时，为了让重要消息首先发送，给不同类型的消息定义了不同的优先级，插入队列时将根据优先级决定它在队列中的位置。API app\_msg\_prio (struct app\_msg \*msg)根据消息的命令字返回其对应的优先级数值。将新消息插入队列时，从队列头开始逐项与新消息比较优先级，如果新消息优先级大于当前项，将新消息插入到当前项之前，否则比较下一项；如果新消息的优先级小于当前队列中的所有项，则添加到队列尾。通过上述插入方法，高优先级的消息排在队列前部，同优先级的消息根据入队的先后排序、后入队的排在后面。

消息队列需要用到以下结构体：

#define APP\_MSGMEM\_LEN 6

struct msg\_entry {

struct msg\_entry \*next; // 指向下一个消息项的指针

struct app\_msg msg; // 消息数据

rimeaddr\_t to; // 消息的目标地址

uint8\_t cmdop; // 消息的类型

uint8\_t len; // msg域的实际长度

uint8\_t retry; // 最大重传次数

uint8\_t ack; // 是否要等待AP的ACK

uint8\_t prio; // 消息的优先级

int index; // 消息在flash中的索引

};

MEMB(app\_msg\_mem, struct msg\_entry, APP\_MSGMEM\_LEN);

LIST(app\_send\_queue);

MEMB()宏定义了一块RAM区域app\_msg\_mem，通过memb\_alloc()可以申请一块struct msg\_entry大小的RAM并获得它的指针，使用该指针可以填入消息内容、消息优先级等信息，再将该指针加入app\_send\_queue的队列链表中。根据队列操作时，从app\_send\_queue取得队列头的消息项指针，调用相关API以该指针为参数进行消息发送等操作。对某消息项的操作完成后，需要从app\_send\_queue中删除该项，并且调用memb\_free()将该项对应的RAM归还给app\_msg\_mem用于后续使用。

## RP节点主程序

apps/wvds/router/router.c为RP节点的主线程文件。相对于VD和AP，RP的主线程功能较为简单，主要包括启动相关的子线程、建立与AP间的无线连接、启动多个软件定时器执行心跳等任务、网络层组件转发需要中转的消息等。RP主程序的主要流程见图 11。

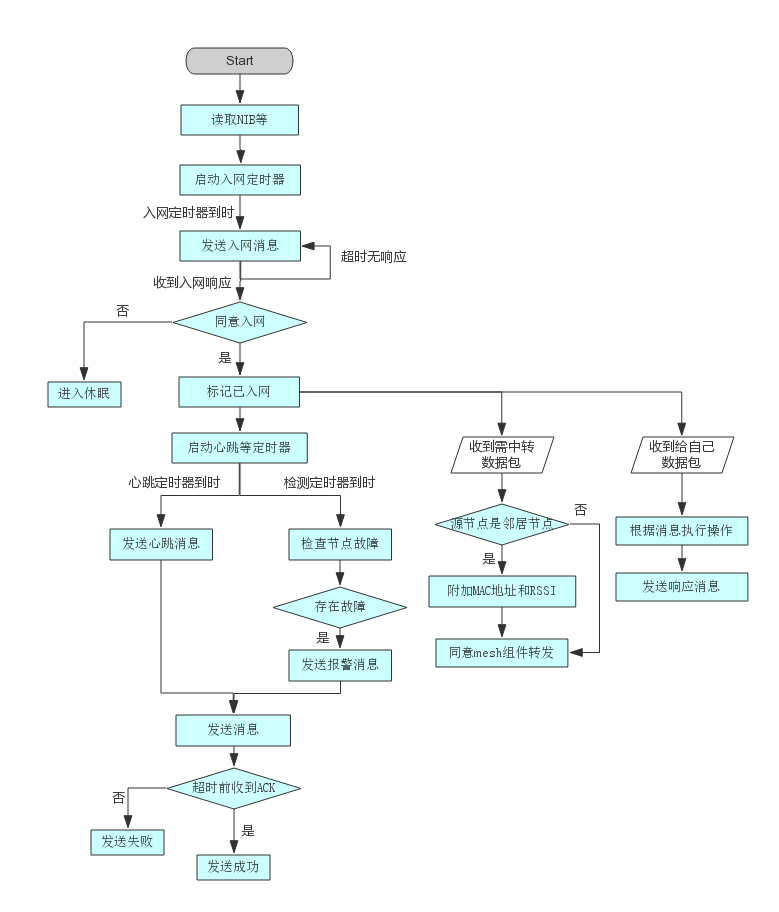


图 11 RP主线程流程

## AP节点主程序

apps/wvds/gateway/gateway.c为AP节点的主线程文件，它的主要功能包括启动相关的子线程、控制GPRS模块建立与服务器间的TCP连接、接收来自服务器的数据并做相应处理、接收发送自子网VD/RP的数据并做相应处理、启动多个软件定时器执行相应任务。AP主程序的主要流程见图 12。

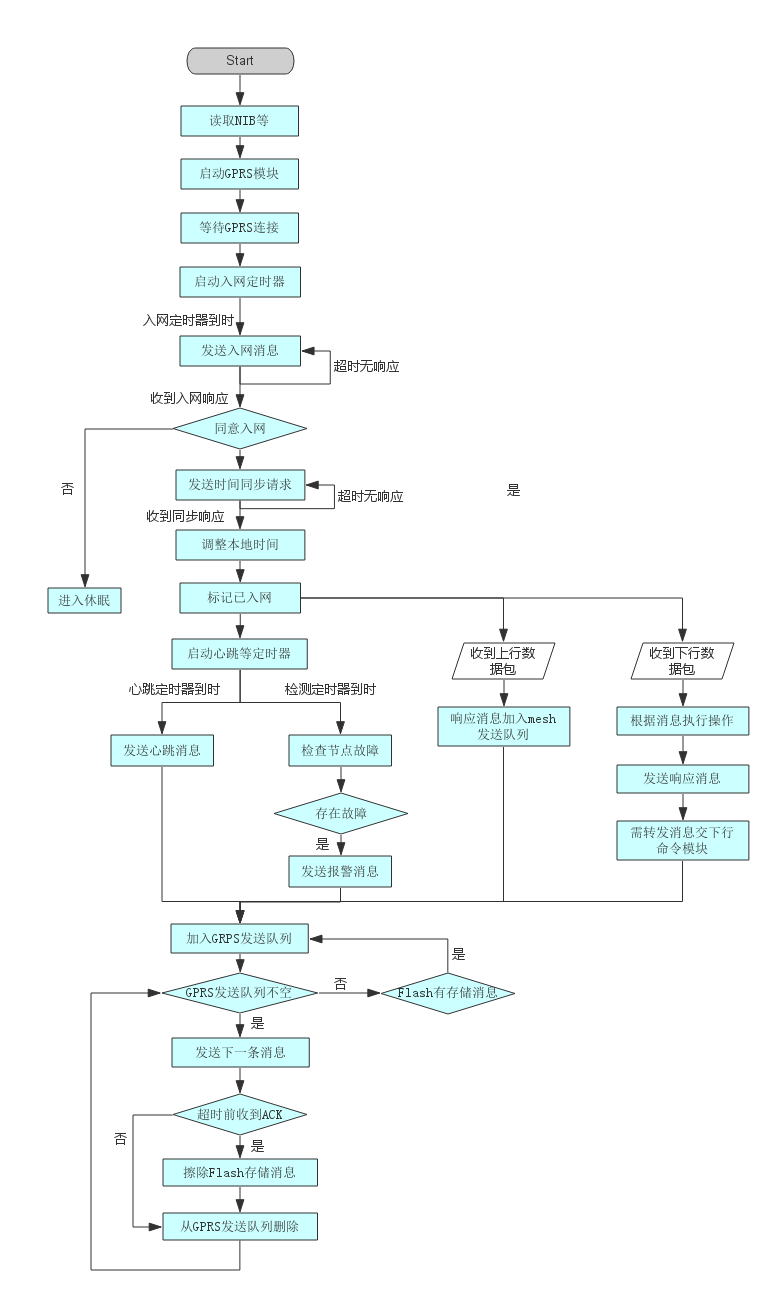


图 12 AP主线程流程

### 关键消息数据存储

与VD主程序类似，AP主程序对接收到的关键消息也会先存储到Flash存储器后再发送，同样采用将Flash空间作为环形队列使用的方式。涉及到的API包括：

int app\_save\_msg(struct app\_msg \*msg, uint8\_t len);

int app\_load\_msg(uint8\_t \*buf, uint8\_t \*len);

int app\_erase\_msg(uint16\_t idx);

void app\_send\_saved\_msg(void);

与VD主程序的不同之处在于，AP存储消息时保存了整个struct app\_msg，而VD只保存了struct app\_msg中的cmdop域和有效数据data域。

### 支持优先级的消息队列

与VD主程序类似，AP主程序也使用支持优先级的消息队列来存储要向服务器发送的消息。当AP要主动向服务器发送消息、或者AP要转发从RP/VD收到的消息给服务器时，调用app\_msg\_prio(struct app\_msg \*)可以得到该消息对应的优先级，再调用app\_gprs\_enq()将消息加入GPRS发送队列。与VD的队列相同，高优先级的消息排在队列的前部，同优先级的消息根据加入队列的先后排序、后入队的排在后面。

### 共享消息缓存和多消息队列

AP主程序要处理消息的情况相对复杂一些。AP在接收到网内RP/VD的上行数据包时，有的需要通过GPRS转发到服务器，有的需要AP作相应处理并向其发送对应的响应消息，有的两者都需要。AP在接收到来自服务器的下行数据时，有的需要转发到网内对应的目标RP/VD节点，有的需要作处理后向服务器发送响应消息，有的两者都需要。此外，在AP主线程的某API中接收到消息时，已经有较多层的函数调用，在其中再直接进行数据发送可能导致函数调用的层次过多而出现堆栈溢出问题。

因此，AP在处理消息时设置了共享的消息缓存和多个消息队列，对应于gateway.c代码中的以下几个结构体：

#define APP\_RECVQ\_LEN 4

#define APP\_SENDQ\_LEN 4

#define APP\_GSENDQ\_LEN 5

#define APP\_GRECVQ\_LEN 3

#define APP\_MSGMEM\_LEN (APP\_RECVQ\_LEN + APP\_SENDQ\_LEN + APP\_GSENDQ\_LEN + APP\_GRECVQ\_LEN)

struct msg\_entry {

struct msg\_entry \*next; // 指向下一个消息项

uint8\_t act; // 消息需进行的操作

struct app\_msg msg; // 消息数据

rimeaddr\_t from; // 消息的源节点地址

uint8\_t hops; // 消息的源节点距AP跳数

rimeaddr\_t to; // 消息的目的地址

uint8\_t len; // msg域的实际长度

uint8\_t retry; // 发送时的最大重传次数

uint8\_t prio; // 消息的优先级

int index; // 消息在Flash中的存储索引

};

MEMB(app\_msg\_mem, struct msg\_entry, APP\_MSGMEM\_LEN);

LIST(app\_recv\_queue);

LIST(app\_send\_queue);

LIST(app\_gprs\_sendq);

LIST(app\_gprs\_recvq);

enum {

ACT\_RECV = 0x01,

ACT\_SEND = 0x02,

ACT\_GPRS\_RECV = 0x04,

ACT\_GPRS\_SEND = 0x08

};

其中app\_msg\_mem是各消息队列共享的消息缓存，app\_recv\_queue是从网内RP/VD接收消息的队列，app\_send\_queue是向网内RP/VD发送消息的队列，app\_gprs\_sendq是向服务器发送消息的队列，app\_gprs\_recvq是从服务器接收消息的队列。同时定义了几个枚举值表示要对消息项进行的处理操作，它们各自只占用一个不重叠的比特位，因此可以组合表示该项需要进行多个操作。

以车位检测器事件消息0x01为例，在Rime协议栈的底层接收到该消息时，逐层向上调用，最终调用AP主线程gateway\_process中的app\_mesh\_recv()来处理收到的消息。在app\_mesh\_recv()中，将调用memb\_alloc(app\_msg\_mem)申请一个消息项，把消息的数据内容和相关信息如源节点地址、源节点跳数等存入该消息项，置域act的ACT\_RECV对应位为1，再将该消息项指针加入到app\_recv\_queue链表，调用process\_post(&gateway\_process, mesh\_rcvd\_event, NULL)让AP主线程稍后再处理该消息。在系统调度器根据mesh\_rcvd\_event事件call\_process到gateway\_process时，从app\_recv\_queue队列头中取出上述0x01消息对应的消息项，该消息要求给对应的VD发送一个确认消息，因此又从app\_msg\_mem申请一个消息项并填充对应的确认消息内容并加入app\_send\_queue待稍后发送给该VD；另一方面，该消息需要转发给服务器，因此将该消息项的act域的ACT\_GRPS\_SEND位置为1，并将消息项指针加入到app\_gprs\_sendq待稍后发送到服务器。在以上接收处理操作完成后，需要将该消息项的act域的ACT\_RECV位清0，并从app\_recv\_queue中删除该项。如果某消息项的act域在某个位清0操作后等于0，则需要调用memb\_free()将其归还给app\_msg\_mem供后续使用。在上述例子中，在接收处理完成后因为act域的ACT\_GPRS\_SEND位为1，还不能立即归还，需要在GPRS发送操作完成后ACT\_GPRS\_SEND位也清0后才能归还。

### 关键数据作为历史数据发送

泊位事件消息0x01和磁场数据消息0x24是本系统的两种关键数据消息，如果AP可与服务器正常通讯，则这些消息立即发送给服务器，否则它们会被存储在外部Flash中，待AP恢复与服务器的通讯后再发送，有些会作为历史数据消息发送。

由于磁场数据消息0x24本身长度就较大，且数据内容不便于合并，该类型不作为历史数据发送。历史数据消息0x23是将多个泊位事件消息0x01合并为一条消息，它对应的数据结构定义如下：

#define APPDATA\_HISTDATA\_NUM 5

struct app\_data\_histdata {

uint8\_t tstamp[TSTAMP\_LEN]; // 本条消息生成的时间戳

uint8\_t count; // entries中实际有效数据的个数

uint8\_t reserv;

struct {

uint8\_t devmac[DEVMAC\_LEN]; // 当前事件项的设备编码

uint8\_t tstamp[TSTAMP\_LEN]; // 当前事件项的时间戳

uint8\_t status; // 当前事件项对应的有无车状态

} entries[APPDATA\_HISTDATA\_NUM];

};

受本系统数据帧长度和AP所用RAM的限制，目前限定每条历史数据消息最多包括5条事件数据消息。

为了支持历史数据机制，在存储消息的元数据中增加了一个域，即struct msgmeta中的history域，当该值为1时，表示是历史数据，为0时表示是实时数据。目前AP从VD接收到事件消息时，首先要存储到外部Flash中，如果当时AP与服务器通信正常，作为实时数据保存，如果AP未能与服务器连接，则作为历史数据保存。另外，当一条事件消息作为实时数据由AP向服务器发送多次都未收到ACK时，也标记该条数据为历史数据。

上述说明的一个要点就是，在AP无法与服务器通信期间，接收到的数据保存到外部Flash时都作为历史数据。在AP恢复与服务器的通信后，这些历史数据中有一些应当再标记为实时数据上传，因为实时数据和历史数据在服务端的处理策略不同，服务端需要这些实时数据处理AP重新上线后相应泊位的状态变化和相关收费问题。AP重新上线后，将调用app\_remark\_saved\_msgs()重新标记一些重要的历史数据为实时数据，要点如下：

1. 使用双重循环检查历史数据消息，外层循环根据消息找到一个尚未完成标记处理的VD设备，内层循环根据外层确定的VD设备编码扫描确定其最后一条已上传的消息；
2. 外层循环从外部Flash消息循环队列的尾部向头部扫描。该队列的尾部总是存储着发生时间更晚的事件消息，即离当前时刻的时间更近；
3. 使用一个cache记录某个泊位是否已扫描完成，如果该cache中已存在某个泊位VD的设备编码，则该泊位已扫描过，跳过当前消息，继续检查前一条消息；
4. 确定一个未完成标记的VD后，内层循环开始，从外层循环指针的当前位置继续向队列头部扫描，还会越过头部扫描之前已上传的消息，内层循环在找到了该VD最后一条已上传的消息后结束，同时会记录下该条消息之后的消息对应的索引；
5. 外层循环的当前消息索引记为latest，内层循环确定的最后一条已上传消息的索引记为lastup，lastup的下一条该VD消息的索引记为lastnext。如果因为消息覆盖导致某VD找不到最后一条已上传消息，即lastup、lastnext不存在，则该VD所有消息仍作为历史数据上传；如果latest等于lastnext，则表示该VD只有一条消息未上传，标记latest为实时数据上传；如果latest大于lastnext，表示该VD至少有两条消息未上传，进一步，如果lastup消息是有车且lastnext消息是无车，则lastnext消息需要标记为实时数据，它上传后用于结束掉线前最后一辆车的计费，而latest消息如果是有车的话，总是应该标记为实时数据上传，开始AP重新上线前已驶入车的计费。

当AP与服务器通信正常时，AP首先发送GPRS发送队列中的消息；当队列中已没有要发送的消息时，扫描外部Flash找到还未上传的实时泊位事件或磁场数据消息，加入GPRS发送队列发送；如果在外部Flash中也找不到实时泊位事件和磁场数据消息，则会调用app\_load\_hist\_msg()扫描找出历史数据合并为历史数据0x23消息加入GPRS发送队列发送。AP扫描历史数据构造0x23消息时从队列头部向尾部扫描，在最后可能出现合并消息只有少于最大数量的事件数据。AP向服务器发送实时数据包括重传失败后标记其为历史数据，而发送历史数据失败后将调用app\_erase\_hist\_msg()擦除该合并消息对应的多条事件消息。

# 各功能模块流程图和函数的输入输出说明

## 信息存储模块

节点运行时需要一些使用配置数据，为实现节点重启后仍然能够使用，这些配置数据需要保存到外部Flash存储器中。下表给出了VD节点外部Flash的空间分配计划：

|  |  |  |  |
| --- | --- | --- | --- |
| 数据名称 | 起始地址 | 大小(B) | 备注 |
| struct OTA | 0x00000 | 64 | 无线升级信息，包括自身程序版本、缓存程序版本等 |
| struct PIB | 0x00040 | 64 | 全网参数信息，包括心跳周期、报警阈值等 |
| struct NIB | 0x00080 | 128 | 节点特定信息，包括设备编码、AES密钥等 |
| struct STATS | 0x00100 | 128 | 节点统计信息，包括无线收发次数、能耗数据等 |
| struct ALGO | 0x00180 | 128 | 检测算法参数，包括采样周期、检测阈值等 |
| Route Table | 0x00200 | 1024 | 路由表信息，每项4字节，最多255项 |
| Nodes Table | 0x00600 | 2048 | 网内节点信息，仅AP使用，每项8字节，最多255项 |
| Nodes Blacklist | 0x00E00 | 512 | 节点黑名单，仅AP使用，每项8字节，最多64项 |
| struct MSGS | 0x01000 | 32kB | 关键消息数据，VD和AP使用 |
| CFS metadata | 0x09000 | 2048 | CFS文件系统元数据 |
| Update Image | 0x09800 | 90kB | VD升级固件，用于保存无线升级时接收的新固件数据 |

表 9 VD外部存储器空间分配表

在VD/RP/AP主线程启动后，首先要读取上表中定义的一些配置数据。在apps/wvds的公用文件app.c中提供了API nv\_write()、nv\_read()对这些配置数据进行读写。下面以节点特定信息struct NIB进行说明：

struct NIB {

uint16\_t magic;

uint8\_t devmac[6]; // 设备编码

uint16\_t addr; // 网内短地址

uint8\_t hwver; // 硬件版本号

uint8\_t fwver; // 固件版本号

uint8\_t activated; // 激活标志

uint8\_t tricseq; // 全网参数序号

uint8\_t locked; // 网络锁定标记(仅AP使用)

uint8\_t reserved[1];

uint8\_t aeskey[16]; // AES加密密钥

uint8\_t aesiv[16]; // AES加密向量

char host[16]; // 服务器IP

uint16\_t port; // 服务器端口

uint8\_t reservedb[12];

uint16\_t crc; // NIB的CRC16检验和

};

在VD主线程device.c的app\_init()子函数中有如下代码段：

nv\_read(NV\_NIB\_ADDR, (uint8\_t\*)&nib, sizeof(struct NIB));

ccrc = crc16\_data((uint8\_t\*)&nib, sizeof(struct NIB)-2, CRC\_INIT);

if ((nib.magic != NV\_MAGIC-1) || (ccrc != nib.crc) || (nib.devmac[0] != NODE\_VD)) {

PRINTF("app init nib\n");

memset(&nib, 0, sizeof(struct NIB));

nib.magic = NV\_MAGIC;

nib.devmac[0] = NODE\_VD; // VD(不可更改)

nib.devmac[1] = 0xCA; // 厂商代码，CA拟为Cadre简称

nib.devmac[2] = 0x16; // 生产年份(BCD)

nib.devmac[3] = 0x11; // 生产月份(BCD)

nib.devmac[4] = 0x00; // 生产序号高字节(BCD)

nib.devmac[5] = node\_id; // 生产序号低字节(BCD)

nib.addr = 0x200 + nib.devmac[5]; // 默认短地址(不要更改本行)

nib.hwver = 0x10; // 硬件版本号(BCD), v1.0

nib.fwver = 0x10; // 固件版本号(BCD), v1.0

nib.activated = 1; // 激活标志, 0:未激活, 1:已激活

nib.tricseq = 0; // 全网参数序号

nib.crc = crc16\_data((uint8\_t\*)&nib, sizeof(struct NIB)-2, CRC\_INIT);

nv\_write(NV\_NIB\_ADDR, (uint8\_t\*)&nib, sizeof(struct NIB));

}

根据如上的代码段，首先读取地址NV\_NIB\_ADDR（即表4中的起始地址0x00080）处的Flash数据到结构体struct NIB nib，如果nib.magic等于常数NV\_MAGIC，表示不是第一次上电运行，可以直接使用nib的相关字段配置数据；如果nib.magic不等于常数NV\_MAGIC，表示是第一次上电运行，则需要对nib的相关字段填写默认值，并写入Flash表明已经进行了初始化。

相比于之前的实现，将类似struct NIB的存储结构体最后2字节定义为16位校验和CRC，用于校验结构体中除CRC外的数据的正确性，避免Flash数据被意外改变时无法发现而导致设备程序无法正常运行。读取数据时一并读取CRC，将结构体其他部分计算CRC并与读取的CRC进行比较，如果两者不一致，则使用默认值重新初始化这些结构体。保持CRC在结构体尾部的另一个用处是如果新版本程序相对于旧程序更改了结构体的定义，则新程序在节点上初次运行时几乎不可能出现CRC匹配的情况，因此会重新进行初始化。需注意，改变结构体某个成员的值并再写入Flash时，需要重新计算CRC并赋值后再进行Flash写操作。CRC的计算采用本系统统一使用的CRC16计算法，对应CCITT标准、多项式X^16+X^12+X^5+1、计算初始值是0。

## 节点激活模块

### VD节点激活

节点激活模块负责实现VD节点从部署前休眠状态转换到正常工作状态。VD节点由于生产后需要密封，在运输到部署现场进行安装之前需要保持低功耗模式。在安装到施工现场的某车位后，要按照一定的步骤激活该VD设备。本系统目前采用强磁激活方式。

为了避免设备生产、系统测试、系统部署过程的意外影响，应定义几个固定信道：1、现场信道，即系统在现场运行时一般使用的信道；2、激活信道，设备出厂时默认的信道，与手持激活设备配合的激活节点工作在该信道；3、测试信道，实验室研发测试时用到的所有节点工作在该信道。

如下的顺序图图 13描述了VD节点激活过程的流程，其中VD表示VD节点、ACT表示激活节点、APP表示安装了定制的激活App的手持设备。激活流程的要点包括：

1. 生产后部署前VD处于休眠模式，关闭射频，磁场采样和强磁检测功能正常运行；
2. VD设备受到磁干扰，采样并计算后算法模块判断为是强磁信号，则VD开启射频并主动周期性发送激活请求0x33消息；由于是干扰，一段时间如2分钟后因为不会接收到来自激活器的响应消息，VD关闭射频返回休眠模式；
3. 现场安装后，将激活器放置到VD设备上方，算法模块检测出强磁信号，VD开启射频并主动周期性发送激活请求0x33消息；与手持设备配合的激活节点接收到该请求消息，通知手持设备App有节点要激活，向VD发送0x33响应消息；VD接收到响应消息，保持射频开启，启动一个超时定时器，等待App发起的标定等相关操作；
4. VD设备接收到激活App发起由激活节点发送的标定等操作消息时，根据消息的操作命令字和参数执行相应操作，并重新启动超时定时器；
5. 激活App完成所有激活过程要求的操作后，向VD设备发送断开连接0x35消息，消息中包括现场信道的信道号；VD设备接收到该消息时，关闭射频并停止超时定时器，开始在现场信道上正常工作；如果VD设备超时未收到断开连接0x35消息，则关闭射频返回休眠模式。

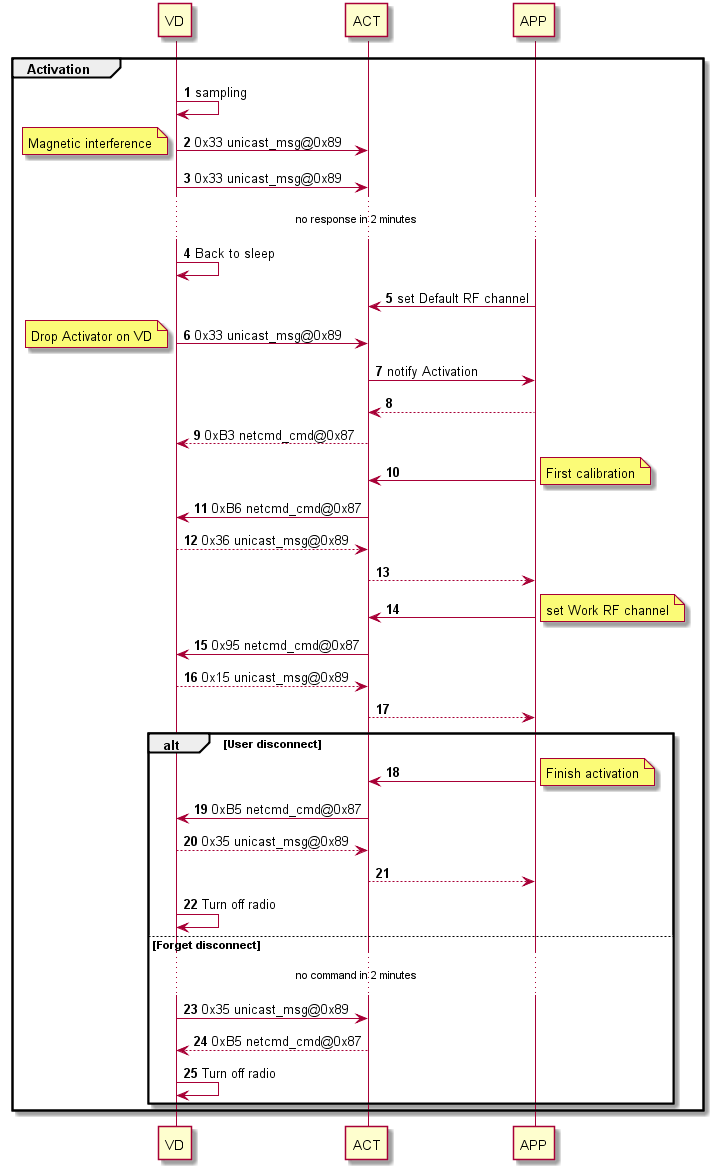


图 13 VD首次激活交互顺序图

### AP和RP激活

为了简化AP和RP出厂后在现场安装时的设置步骤，定义该过程为AP和RP的激活，目前仅涉及设置工作射频信道。

具体来说，AP和RP使用NIB中的activated域记录是否已激活。出厂烧录后NIB.activated为0，表示未激活，设备上电后提示用户未做初始设置并等待用户通过按键进行设置；通过板上按键设置射频信道后，再按某键将保存信道到外部Flash、修改NIB.activated为1并保存到外部Flash。之后设备重启的话因为NIB.activated为1将跳过初始设置的等待而直接运行设备程序。

AP和RP使用4个板上LED指示重要的运行状态。设备未激活时，上电或重启后所有指示灯闪烁3次提示未激活，然后按二进制形式以亮灭显示设备的当前信道。按照OD4/GD3/YD2/RD1的顺序，分别对应信道值的二进制表示的低4位，1对应亮、0对应灭。比如信道6，二进制表示为0000 0110，低4位是0110，所以OD4灭、GD3亮、YD2亮、RD1灭。设备激活后，RD1指示设备工作状态，正常时每1秒闪烁一次；YD2指示设备入网状态，灯亮时表示尚未成功入网，入网后指示灯灭；GD3指示射频发送，每发送一个数据包该指示灯改变状态；OD4指示射频接收，每接收到一个数据包该指示灯改变状态。

设备按键S2用于切换设备的射频信道。设备未激活时，每按一次S2键，信道切换到下一个。按照802.15.4标准在915M段的信道分配，可用信道是1到10；设备出厂时默认是信道2，按一次S2键后改为3，再按后改为4；当前信道是10时，按S2键后改为1。按S2键成功改变信道后，板上4个LED的显示也随之改变；有时按键后可能出现设置失败，这时需要再次进行按键操作。

设备按键S1用于切换设备的激活状态。设备未激活时，按S2键设置好设备的射频信道后，再短按键S1将激活设备，即把射频信道PIB.radioChan和激活标志NIB.activated写入外部Flash并开始运行设备主程序功能；设备已激活后，按S1键将使设备恢复出厂设置，即NIB、PIB、路由表和文件表等外部Flash中信息将恢复到出厂时的状态，射频信道PIB.radioChan恢复为2、激活标志NIB.activated恢复为0；恢复过程需几十秒，期间RD1指示灯停止闪烁，完成后设备将自动重启，将看到4个指示灯同时闪烁表明已返回未激活状态。

## AT指令模块

本模块实现对AT指令的支持，可以提供简单的操作接口对设备的设备编码、信道、功率、服务器地址等信息进行查询或设置，对应的代码在apps\at-cmd\at-cmd.c中。

AT指令的命令帧以AT+开始、以\r\n结束，没有响应数据时，设备执行成功时返回AT+OK\r\n，有错误时返回AT+ERRx\r\n，其中x是错误代码；有响应数据时，设备返回响应数据的ASCII形式，以\r\n结束。

AT指令模块提供API atcmd\_register(const char \*cmd, atcmd\_callback\_t cb)使调用模块可以方便地自定义指令，例如atcmd\_register("CHAN=", app\_set\_channel) 将定义一条指令AT+CHAN=6\r\n，对应的处理函数是app\_set\_channel(const char \*arg, int len)，其中arg是指令的参数部分、len是参数部分的字节个数。

AP和RP支持表 10列出的AT指令：

|  |  |  |  |
| --- | --- | --- | --- |
| AT指令名 | 示例 | 说明 | 执行成功响应 |
| MAC? | AT+MAC?\r\n | 查询设备编码 | 04CA17040001\r\n |
| MAC= | AT+MAC=04CA17040002\r\n | 设置设备编码 | AT+OK\r\n |
| CHAN? | AT+CHAN?\r\n | 查询射频信道 | 2\r\n |
| CHAN= | AT+CHAN=6\r\n | 设置射频信道 | AT+OK\r\n |
| POWER? | AT+POWER?\r\n | 查询射频功率 | 24\r\n |
| POWER= | AT+POWER=27\r\n | 设置射频功率 | AT+OK\r\n |
| SERVER? | AT+SERVER?\r\n | 查询服务器地址 | 58.250.57.68:4030\r\n |
| SERVER= | AT+SERVER=58.250.57.68:4033\r\n | 设置服务器地址 | AT+OK\r\n |
| FRESET | AT+FRESET\r\n | 恢复出厂设置 | AT+OK\r\n |
| REBOOT | AT+REBOOT\r\n | 重启设备 | AT+OK\r\n |

表 10 AT指令列表

## 节点入网模块

节点入网的流程在AP/RP/VD的主线程中实现，分别在文件gateway.c、router.c、device.c中有相关的代码。目前的入网机制只支持部署前该子网的所有节点已设置了固定的射频工作信道，现场部署时如果同一区域要部署两个子网，则它们出厂时需设置好相应的射频工作信道。如果需要在现场自动形成两个子网，后续需增加信道扫描等支持。

某个子网内的节点间使用16位的短地址进行通讯，目前系统支持一个子网的网络容量是255个节点，因此要求所有节点的短地址小于256。AP节点总是使用短地址1，RP/VD节点的短地址在入网时由所属AP进行动态分配。AP接收到RP/VD入网消息时如果短地址超过合法范围（即大于256）或者与另一个节点的短地址冲突，则AP给该节点分配一个新的短地址。AP将节点的64位MAC地址和16位短地址的对应关系保存到外存储器的“网内节点信息”段，以短地址为索引计算地址后写入MAC地址，例如节点01CA16100003的短地址是4，则写入地址为NV\_NODES\_ADDR+8\*4。RP/VD会保存AP分配的短地址到外存储器，后续节点重新启动后就直接使用该短地址，入网时一般也无需由AP再次分配短地址。RP/VD第一次上电运行时会初始化外存储器中的短地址，目前总是初始化为一个大于256的值，因此肯定会由AP分配新的短地址。

### AP节点入网

AP节点上电以后，主线程启动，按照以下步骤进行入网：

1. 标记进入START状态，读取NIB等配置数据；
2. 初始化和启动子网内通信模块；
3. 启动GRPS通信模块，根据NIB中的服务器IP和端口连接服务器，主线程等待连接完成；
4. GPRS通信模块发送一系列指令连接服务器，成功连接后通知主线程；
5. 主线程收到GRPS模块连接成功的通知，标记进入JOIN状态，启动入网定时器；
6. 入网定时器到时，发送入网消息0x11，设置响应消息超时；
7. 超时前接收到入网响应消息，转第8步，未收到响应，再启动入网定时器，转第6步；
8. 入网响应同意入网，转第9步，不同意入网，转第13步；
9. 标记进入SYNC状态，停止入网定时器、启动时间同步定时器；
10. 时间同步定时器到时，发送时间同步消息0x05，设置响应时间超时；
11. 超时前接收到同步响应消息，转第12步，未收到响应，再启动时间同步定时器，转第10步；
12. 根据响应消息中时间戳设置本地时间，标记已成功入网、进入READY状态，转第14步；
13. 标记进入SLEEP状态，停止入网定时器。
14. 启动周期性的心跳定时器、故障检查定时器，进入常态运行。

AP节点还需要按照以下流程处理VD/RP节点的入网消息，相关的流程主要在gateway.c文件的app\_handle\_join()子函数中：

1. 查找黑名单中是否有当前消息源节点的MAC地址，如果有，置reject标记为1，转第9步，否则转第2步；
2. 查找Flash中已入网节点列表中是否有当前消息源节点的MAC地址，如果有，转第3步，否则转第4步；
3. 比较当前消息的源节点短地址与已入网节点列表中对应的短地址是否一致，如果是，置accept标记为1，转第8步，否则置assign标记为1，转第10步；
4. 查询服务器已认证节点列表中是否有源节点的MAC地址，如果没有，通过GPRS向服务器转发本消息，否则转第5步；
5. 检查源节点的短地址是否大于最大合法值，如果是，转第7步，否则转第6步；
6. 查询Flash中已入网节点列表，检查源节点短地址是否对应其他节点，如果是，表明有短地址冲突，转第7步，否则置accept为1，将源节点MAC地址存储Flash，转第8步；
7. 扫描Flash中已入网节点列表，找到一个合法的尚未被使用的短地址，置assign为1，转第10步；
8. 发送响应消息给源节点同意其加入本子网；
9. 发送响应消息给源节点拒绝其加入本子网；
10. 发送响应消息给源节点让其设置新的短地址后再尝试入网；

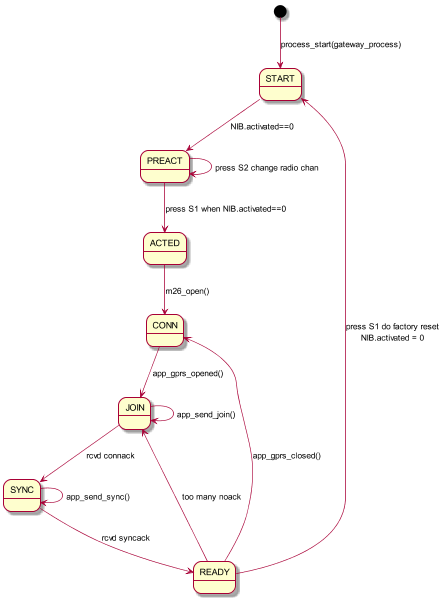


图 14 AP节点主程序状态图

### RP节点入网

RP节点上电以后，主线程启动，按照以下步骤进行入网：

1. 标记进入START状态，读取NIB等配置数据；
2. 标记进入JOIN状态，启动入网定时器；
3. 入网定时器到时，发送入网消息0x21，设置响应消息超时；
4. 超时前接收到入网响应，转第8步，未收到响应，再启动入网定时器，转第6步；
5. 入网响应同意入网，转第6步，不同意入网，转第7步，分配了新地址，转第8步；
6. 根据响应时间戳更新本地时间，停止入网定时器，标记已入网、进入READY状态，转第9步；
7. 停止入网定时器，关闭射频，标记进入SLEEP状态；
8. 根据响应内新地址设置节点短地址，写入NIB，转第3步；
9. 启动周期性的心跳定时器、故障检查定时器，进入常态运行。

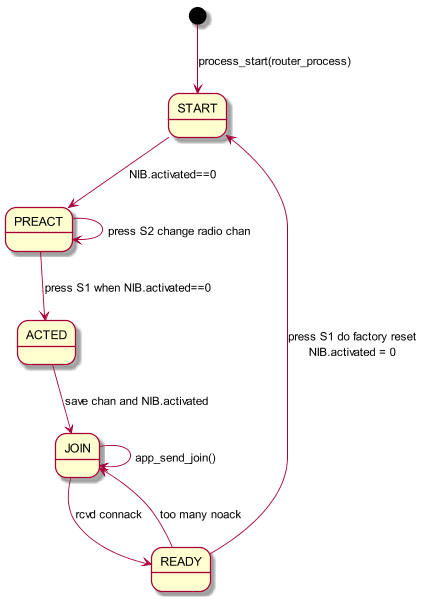


图 15 RP节点主程序状态图

### VD节点入网

VD节点上电以后，主线程启动，按照以下步骤进行入网：

1. 标记进入START状态，读取NIB等配置数据，启动周期性磁采样；
2. 根据nib.activated判断节点是否已被激活，如果已激活，标记进入JOIN状态，poll主线程，转第5步，否则转第3步；
3. 采样定时器到时后采样磁信号，交给算法模块判定，为激活信号时标记进入ACTED状态，poll主线程；
4. 主线程处理POLL事件，为ACTED状态，标记进入JOIN状态，poll主线程；
5. 主线程处理POLL事件，为JOIN状态，启动入网定时器；
6. 入网定时器到时，发送入网消息0x21，设置响应消息超时；
7. 超时前接收到入网响应，转第8步，未收到响应，再启动入网定时器，转第6步；
8. 入网响应同意入网，转第9步，不同意入网，转第10步，分配了新地址，转第11步；
9. 根据响应时间戳更新本地时间，停止入网定时器，标记已入网、进入READY状态，转第12步；
10. 停止采样及入网定时器，关闭射频，标记进入SLEEP状态；
11. 根据响应内新地址设置节点短地址，写入NIB，转第6步；
12. 启动周期性的心跳定时器、故障检查定时器，进入常态运行。

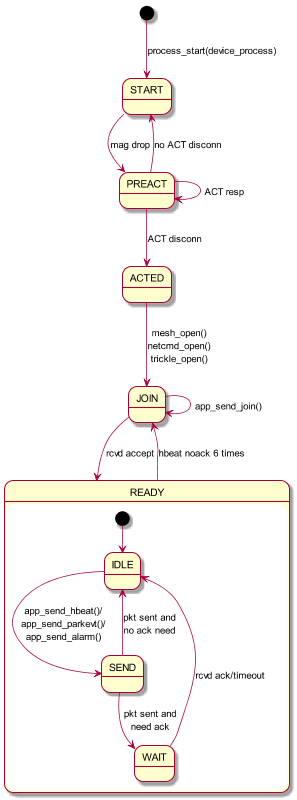


图 16 VD节点主程序状态图

## 心跳数据模块

心跳数据模块负责周期性向服务器发送一个心跳消息，报告当前的泊位状态、磁场数据、电池电压等数据。该模块均在VD/RP/AP的主线程文件中实现。

心跳数据模块使用一个etimer实现周期性发送心跳消息，它的周期由struct PIB中对应的参数项确定。VD/RP/AP节点采用不同的心跳发送周期，分别对应struct PIB中的vdHbeatT、rpHbeatT和apHbeatT。主线程在节点成功加入网络后启动心跳定时器，在心跳定时器到时后，设置心跳定时器的下一次到时时间，设置最大心跳发送次数并调用app\_hbeat\_task()进行心跳消息发送，周期性重复该操作。

在app\_hbeat\_task()中，如果尚未达到最大心跳发送次数，调用app\_send\_hbeat()进行心跳消息的发送，同时启动一个ctimer在10秒后进行下一次发送。如果在ctimer的超时时间内接收到来自服务器或AP的心跳确认消息，则可判定网络正常，停止ctimer无需进行下一次发送；否则，ctimer超时，将调用app\_hbeat\_task()再进行一次心跳消息发送。

根据以上机制，以VD节点和30分钟心跳周期为例，VD每隔30分钟进行心跳消息发送，如果网络正常，VD只需发送一次心跳消息就会收到AP对应的确认，否则VD会在发送后10秒再次进行发送直到收到确认消息或达到最大发送次数。

所有VD和RP的心跳发送时刻是不同步的，根据该节点的启动时刻和加入网络的时刻决定，较大的心跳周期和不太多的子网内节点数量使得心跳发生冲突的概率不大。

## Mesh网络模块

Mesh网络模块负责实现子网内任意两个节点间的通信。实际使用中，VD/RP的上行数据只会发送到AP节点，下行数据也只会由AP发送到特定的VD/RP。该模块的实现在文件contiki/core/net/rime/mesh.c中，依赖于Rime协议栈的较多组件。

Mesh网络模块提供的API较为简单，使用时需注意以下几点：

1. mesh\_open()：打开一个mesh连接用于mesh式网络通信。调用时需要提供一个回调函数结构体的指针，该结构体包含多个函数指针，分别对应接收到mesh消息时、mesh消息发送成功后、mesh消息发送失败后、mesh消息转发前的主线程中的处理函数。该API执行时将自动打开依赖的multihop连接和route-discovery连接。
2. mesh\_close()：关闭当前使用的mesh连接，自动关闭依赖的multihop和route-discovery连接。
3. mesh\_send()：在当前的mesh连接上发送数据。使用时调用者应该在前一个数据包发送完成之后再发送下一个数据包，因为mesh模块并不提供队列缓存多个消息。

mesh消息转发前的回调函数可以用于在转发前对数据进行适当的修改，在RP主线程router.c中该回调函数为app\_mesh\_intercept，目前进行两种形式的数据修改：1）向特定数据包插入该RP的MAC地址以提供网络拓扑信息；2）向特定数据包插入接收该数据时RP检测到的信号强度即RSSI以提供信道质量信息。

### Mesh广播

为了优化VD设备的功耗，本系统对mesh模块进行了修改使VD设备按照广播方式发送上行数据，按照该方式，VD广播的上行数据可能被AP和多个RP接收并转发，从而减少VD向特定父节点发送上行数据时因信道质量差或冲突干扰导致的重传和路由调整代价。

修改版的mesh实现定义了预编译符号MESH\_BROADCAST，原本的运行方式对应MESH\_ BROADCAST为0，要采用广播发送方式时，可在主工程目录下的project-conf.h中定义MESH\_ BROADCAST为1。

VD广播发送的数据被多个RP或AP接收时，它们不立即转发该数据，而是设置一个随机的延时后再转发。在延时期间，可能出现两种情况：1、AP已接收到该数据，因此会向源VD发送响应数据，RP监听到该响应数据时将取消对应上行数据的转发；2、RP监听到其他延时较短的RP先转发的数据，这时RP重新设置延时以避免不必要的重传，在重新延时期间可能监听到AP的响应数据按情况1取消转发。

虽然RP按上述措施会一定程度上避免向AP转发重复的VD数据，AP处仍然有可能接收到重复的数据。AP处增加了两个去除重复数据的措施：1、增加了接收cache。每接收到一个数据包，根据源节点MAC和消息类型、时间戳检查cache中是否有该消息，如果有，直接忽略该消息，否则加入接收队列待处理且同时添加到该cache。接收cache存在一个问题是如果源VD/RP因为没有接收到响应消息而重传，AP可能检测为重复消息而跳过再次发送响应消息；2、增加了GPRS转发cache。AP向服务器转发数据时，先根据源节点MAC和消息类型、时间戳检查该cache中是否已有该条数据，如果有，跳过向GPRS转发，否则将该条数据加入GPRS发送队列且同时添加到该转发cache。

## 路由发现模块

路由发现模块负责通过交换路由请求和回复消息找到从源节点到目标节点的传输路径，供mesh或其他模块根据路径进行数据转发。模块实现在文件contiki/core/net/rime/route- discovery.c中。

当mesh或其他模块需要发送数据时，需要首先确定发送给哪一个邻居节点，将调用路由维护模块API route\_lookup(rimeaddr\_t \*dest)查询到达目标节点dest的路由，如果不存在该路由，就需要调用route\_discovery\_discover()向子网内节点查询路由。该模块将调用netflood\_send()通过洪泛方式发送route req消息，接收到route req消息的节点会根据该消息的源节点和发送节点添加到源节点的相应路由项，当接收到该消息的节点发现请求是到自己的路由时，将向源节点发现route rep消息，在多跳的情况下，route rep消息可按照route req消息的反向路径转发到源节点。该模块在接收到route rep消息发现到目标节点的路由时，将调用相关回调函数使mesh等模块开始发送数据；如果超过一定时间仍未发现路由，也会调用另一回调函数通知mesh等模块进行处理。

相对于原版的route-discovery模块，本系统进行了如下的修改：

1. VD节点不转发route req，只回复对节点本身的路由请求。由于VD需要保持低功耗，它不应参与消息转发，如果向其他节点提供了错误的路由信息，将导致网络不通；

## 路由维护模块

路由维护模块负责需要更新路由信息时在路由表中增加/删除/替换路由表项，同时支持将路由表保存到外部Flash存储器及重启后使用存储在Flash中的路由表项。模块实现在文件contiki/core/net/rime/route.c中。

相对于contiki的原版文件，主要增加了将路由表项保存到Flash存储器的支持。定义了结构体struct nv\_route\_entry用于表示存储在Flash中的路由项数据：

struct nv\_route\_entry {

rimeaddr\_t nexthop; // 下一跳节点的短地址

uint8\_t cost; // 该路径的路由代价，目前是跳数

uint8\_t reserved; // 保留

};

在向Flash中写入或读取某目标节点对应的路由项时，根据该节点的短地址计算其在Flash路由表中的地址偏移，再把上述struct nv\_route\_entry的数据写入Flash中该地址或从该地址读取struct nv\_route\_entry大小的数据。例如，AP节点通过路由发现模块得知要到达节点10需要经过节点4，经该路径到达节点10需3跳，则对应的struct nv\_route\_entry为{nexthop={4,0}, cost=3, reserved=0}，写入或读取时的地址偏移是10\*sizeof(struct nv\_route\_entry)=40=0x0028。由于直接使用目标节点的短地址作为它在Flash路由表中的地址索引，无需扫描即可快速查到它对应的路由项。由此带来的一个问题是节点的短地址应该在一定的范围内，该问题在入网时由AP检查和重新分配短地址可以解决。

在支持使用Flash路由表的情况下，原路由维护模块的相关API的实现也有一些更改：

1. route\_init()：该API初始化路由表，在原初始化RAM路由表后，增加了第一次上电运行时初始化Flash路由表，后续运行时从Flash路由表中读取RAM路由表数量的路由表项；
2. route\_lookup()：该API查询某目标节点的路由，增加了如果在RAM路由表中未找到对应的路由项时从Flash路由表中查找；
3. route\_add()：该API向路由表添加一项，增加了如果是完全新的路由项，向Flash路由表也写入该路由项的数据；
4. route\_remove()：该API从路由表中删除一项，增加了从Flash路由表中删除对应的数据；
5. route\_drop()：新增的API，用于实现RAM路由表已经占满的情况下、在新路由项要插入时从RAM路由表删除一个旧的路由项，同时在该路由项有变化时改写Flash路由表中的对应数据。

## 下行命令模块

下行命令模块负责实现由AP发送的下行数据到达指定的VD/RP节点，特别是为适用于射频关闭的VD需要采用一些特别处理。模块的实现在文件contiki/core/net/netcmd.c中。

下行命令模块提供的几个API简单说明如下：

1. netcmd\_open(uint16\_t period, struct netcmd\_callback \*cb)：打开一个netcmd连接用于发送下行数据，需要参数period指定VD轮询是否有其对应数据的周期、以秒为单位，参数cb是回调函数结构的指针，调用者可以通过cb指定接收到下行数据时的处理函数；
2. netcmd\_send(rimeaddr\_t \*to, uint8\_t msgid, uint8\_t \*data, uint8\_t len)：发送下行数据，需要提供的参数包括：to 目标节点的短地址，可以是某个节点的地址，也可以是广播地址（表示目标是所有网络节点）；msgid 消息类型，用于接收到数据时根据该值对数据进行解析；data 下行数据的内容；len 下行数据的长度。
3. netcmd\_set\_period(uint16\_t period)：设置轮询的周期，调用后VD将改变向父节点查询是否有其对应数据的频率。如果period设置为0，则VD停止周期性发送查询。

下行命令模块依赖一个unicast连接进行数据收发，它会使用3种类型的消息：

1. CMD消息，携带真正的下行数据内容；
2. REQ消息，向父节点查询是否有目的为自己的下行数据包，即VD发送；
3. ACK消息，向父节点反馈某个下行数据包已被成功接收。

当有下行数据要发送时，AP主线程调用netcmd\_send()，netcmd线程将该数据包加入到发送队列。在下行转发过程中，当某RP接收到下行数据时，如果消息的目标地址不是RP本身，RP也会将该数据包加入到发送队列。发送队列每项的定义如下：

struct reqcmd\_item {

struct reqcmd\_item \*next;

struct cmd\_item \*cmd;

};

struct req\_item {

struct req\_item \*next;

rimeaddr\_t addr;

uint8\_t count;

struct reqcmd\_item \*cmds;

};

struct cmd\_item {

struct cmd\_item \*next;

struct ack\_item \*acks;

struct netcmd\_msg cmd;

};

struct ack\_item {

struct ack\_item \*next;

rimeaddr\_t addr;

};

其中struct netcmd\_msg cmd保存下行数据的内容，acks保存需要等待其确认的节点列表。

将一个下行数据包加入发送队列时，如果该数据包的目标地址是某个特定节点，acks列表仅填加对应该节点的一个ack\_item，如果目标地址是广播地址，则向acks列表中为每个邻居节点增加一个ack\_item。

netcmd模块在新数据包加入队列后、或者接收到某个节点的REQ消息时将会触发CMD消息的发送：

1) 如果是新数据包加入了队列，在发送队列中查找第一个目标地址是RP节点、或者目标是广播地址且acks中有RP节点的数据包，将数据包向该RP节点发送。

2) 如果是接收到某VD节点的REQ消息，在发送队列中查找第一个目标地址是该VD节点或目标地址是广播地址的数据包，将数据包向该VD节点发送。特别地，如果发送队列中有多个数据包要向该VD发送，需置CMD消息的more标记为1，通知VD保持射频开启等待后续数据包。

在CMD消息发出后，等待接收节点返回的ACK，如果一定时间内未收到ACK，尝试发送下一个消息；如果接收到ACK，将该ACK对应的命令项的acks列表中相应ACK项删除，删除后如果acks列表为空，则该命令项也可从发送队列中删除，之后尝试发送下一个消息。

VD节点按照设定的间隔周期性地向父节点查询是否有目标节点为该VD的数据包，使用一个etimer来实现周期性查询。VD平时要保持射频处于关闭状态，当etimer到时，netcmd模块开启射频，发送REQ消息，并设置一个超时等待CMD消息；如果超时仍未收到CMD消息，则关闭射频，等到下一次再查询下行数据；如果在超时前从父节点收到CMD消息，向父节点发送ACK消息；如果CMD消息中more域为1，则重新设置超时等待CMD消息；最后一个ACK消息发送完成后关闭射频。

### 拦截式下行

上述下行命令机制在最新实现中仅在通过强磁激活VD后才会生效，需要在现场才能进行操作，另一种方式是在心跳/事件消息上行发送时利用射频正在开启状态发送下行命令。

本机制的要点是，VD上行发送心跳或事件消息后，将会等待来自AP的确认消息，等待期间保持射频开启状态。在此期间，如果AP和RP缓存有目标节点是该VD的下行命令或新全网参数，则向VD发送的参数或命令能够被正常接收并执行。

AP和RP为了实现上述功能，需要维护一个邻居VD的列表，列表中每项包含一个trickle计数和netcmd计数。当有新全网参数下发时，对邻居VD列表的每一项，trickle计数都增加2，这样下次在监听到某VD上传心跳或事件消息时，会发送2次全网参数消息使其应用新的全网参数；当有下行指令下发时，如果是目标为某个VD的，邻居VD列表仅对应的一项netcmd计数加1，如果目标是所有VD，则邻居VD列表所有项的netcmd计数加1，在监听到某VD上传心跳或事件消息时，会模拟一个netcmd REQ消息让netcmd下行命令模块向该VD发送对应的命令消息。

## 全网参数模块

全网参数模块负责维护子网内所有节点需要保持一致的参数集合。模块实现在文件contiki/core/net/rime/trickle.c和主线程中。

trickle实现了这样一种机制，当源节点调用trickle\_send()发送一个数据块时，trickle模块会为该数据块设置一个序号，每次调用trickle\_send()该序号加1，并且，源节点从一个最小时间间隔开始，进行一次实际发送后将该间隔翻倍后进行下一次发送，直到达到最大间隔后维持该间隔。当某节点接收到该数据块时，将消息中的序号与节点当前持有的该数据块序号进行比较，如果消息的序号大于当前序号，保存该数据块内容并把自己当作源节点向下游转发；如果消息序号与当前序号相等，忽略该消息；如果消息序号小于当前序号，重置发送间隔为最小值开始向邻居节点转发该数据块。基于trickle机制，网络内所有节点的该数据块都能够在一段时间后保持一致。特别地，新节点在上电加入网络后也可以自动获得最新的数据块，因此该机制特别适合用来维护全网需要保持一致的参数。

VD/RP/AP的主线程分别调用trickle\_open()启动了在同一个channel上的trickle模块，用于维护struct PIB数据块。在调用trickle\_open()时向trickle模块注册了一个回调函数app\_tric\_recv()，当trickle模块接收到新的数据块内容时，才会调用该回调函数通知主线程数据有改变，主线程需要该回调函数中进行处理，主要是检查数据合法性、将新数据写入PIB、根据改变的参数调整任务等。

原版的trickle机制实现适合于射频一直开启的场景，在VD需要绝大部分时间保持射频关闭的情况下，即使邻居节点发送了新的trickle消息，VD节点很大概率无法接收到。因此，本系统内为适应VD节点的需求进行了如下修改：

1. VD节点不进行trickle转发，即它收到新数据块后，不负责转发给邻居节点，减少其发送数据的次数以延长节点生命期；
2. VD节点在达到最大发送间隔后，保持该间隔周期性发送自己的数据块。由于工况应用中全网参数改变的时效性要求不高，可以接受VD在较长时间后才接收到新参数；
3. VD节点的trickle模块平时保持射频关闭，在发送数据块前开启射频，发送完成后等待一小段时间以便接收可能的新参数，接收到新参数或者超时未收到新参数都关闭射频。

一个需要注意的问题是，一个子网的全网参数应该总是由它的AP节点发布的，因此在AP主线程的app\_tric\_recv()函数中，如果因为意外某节点发布了错误的新全网参数，AP接收到该数据时将发现不一致，AP将调用trickle\_send()重新发送正确的全网参数以修正该错误。在同一个信道有两个以上AP的情况下，AP间不同的全网参数将导致按上述修正机制像乒乓球来回一样不断发送冲突的全网参数消息，目前已修正为后启动的AP接受之前已在运行的AP的参数。

## 低功耗控制模块

VD节点最重要的功能需求之一是保持低功耗。该需求的实现不是在单一模块能够实现的，需要在整个协议栈不同层次的组件中增加相关机制的实现。

首先，需要正确控制射频模块，在大部分时间保持射频关闭。但是，射频关闭之后节点无法接收数据，因此又必须有相应的机制支持下行数据的接收。射频控制有以下几个要点：

1. 上行数据的发送集中在VD主线程中进行，平时保持射频关闭，在有数据要发送前，调用API开启射频，进行上行数据的发送，发送完成后设定一个超时时间。如果在超时结束前接收到对应的下行响应数据，可以立即关闭射频，否则在超时结束时关闭射频；
2. 下行命令通过前述下行命令模块进行传输。VD平时保持射频关闭，根据一定的时间间隔周期性开启射频，开启后向父节点发送请求以查询是否有目标地址的下行数据，同时设定超时时间；如果父节点确实缓存了给该VD的下行数据，多个消息会在超时前逐个发送给该VD，VD接收到最后一个数据后关闭射频；如果父节点没有缓存给该VD的下行数据，VD在超时结束时关闭射频。
3. 全网参数通过前述全网参数模块进行维护。该模块也采用类似机制，VD节点以一个较长的间隔发送自己当前持有的参数集合，发送前才开启射频，发送后设置超时，如果邻居RP或AP节点有较新的参数，它们会在超时前向该VD发送新的参数消息，VD接收后即可关闭射频，否则超时结束时VD关闭射频。
4. 以上三个模块对射频的控制可能存在冲突，比如主线程发送上行数据后还在等待响应，此时下行命令模块也向父节点请求数据，主线程等待的响应消息先到达而主线程关闭射频的话，下行命令模块将无法接收到预期的数据。因此，以上三个模块调用主程序共用文件radio.c中的公共API radio\_on()和radio\_off()来控制射频，把射频模块作为共享资源，每个模块持有一把锁，每个模块开启射频时加锁，关闭射频则只能所有模块均解锁时才能进行。如果后续还有模块需要控制射频，也需采用这种方式。
5. 射频部分平均功耗与以上三个模块中的周期参数密切相关，主要是心跳周期、下行命令轮询周期、全网参数轮询周期。心跳周期和全网参数轮询周期都可以较长，例如30分钟，则每次相关数据收发的能耗平均后对平均功耗的影响不大；下行命令轮询周期与系统对向VD操作时可接受的响应时间有关，如果响应时间要求短，这部分的平均功耗会相应增加。

其次，需要优化传感器采样的功耗。本系统使用的传感器目前在采样时整个节点的峰值电流将达到<5毫安，持续时间约15~20毫秒，由于采样频率较高，例如磁场信号平稳时每1秒采样一次、波动时100毫秒采样一次，该部分能耗平均之后将使得平均电流增加几十到过百uA，相对于节点静态电流几uA将成为能耗的最大部分。优化该部分功耗的措施包括：1）减小采样期间MCU的功耗。传感器自身进行采样过程的电流较小，但持续时间较长（6~8ms），MCU在高速时钟驱动下正常运行任务时工作电流可达4~6mA，在采样进行期间MCU正常运行等待结果将明显耗电更多。通过采用中断方式或者定时回调，MCU发送指令让磁传感器开始采样、然后进入休眠模式；磁传感器采样完成准备好数据后，产生硬件中断通知MCU读取数据并处理。通过该方式，采样和计算过程中的大部分时间MCU休眠，将减少该过程的耗电。通过电流仪表测量后可得出该过程的耗电模式如图 17；2）在检测算法允许的情况下降低采样频率，采样频率的改变直接影响检测过程的响应时间。

再次，优化节点的静态功耗，优化节点程序运行依赖的一些动态功耗。静态功耗是指节点在射频/传感器/Flash等外设都进入休眠模式时节点的整体功耗，它需要对各外设进行正确的低功耗设置、MCU各管脚正确设置消除漏电流、MCU进行低功耗模式，与节点的硬件设计也密切相关。

节点程序或者说Contiki系统的正常运行需要依赖底层的定时系统执行相关任务，系统默认采用定时中断的方式，每隔几毫秒从LPMx模式唤醒、更新本地时间、检查是否有定时任务需要运行，这部分功耗平均起来也有较大的增量，图 17反映了定时中断模式下节点的工作电流情况。目前已经修改为不采用定时中断而是根据定时任务进行动态中断的方式。在contiki-conf.h中定义了预编译符号WITH\_DYNAMIC\_TIMEBASE，即是否使用动态时基机制，默认值为0，符合系统原始的工作方式。VD程序需要省电，应该把WITH\_DYNAMIC\_TIMEBASE定义为1，编译时使能相关的代码。

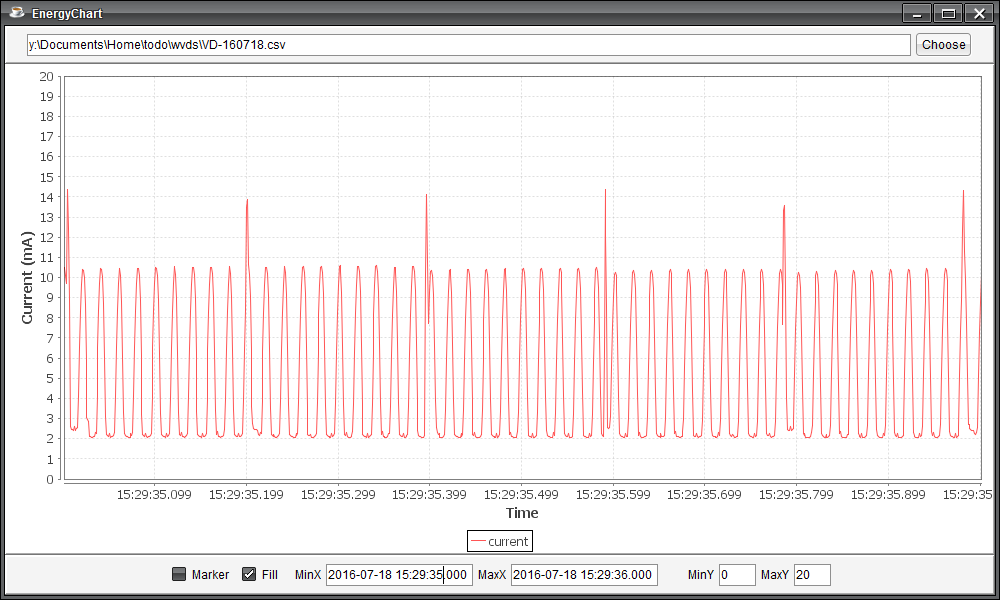


图 17 定时中断电流图

动态时基机制的实现代码主要在平台对应的clock.c和contiki/core/sys/etimer.c中。它的基本思想是不在每个系统节拍的时刻产生中断唤醒MCU，只在定时任务对应的节拍时刻唤醒。etimer.c中有一个链表struct etimer \*timer\_list保存所有的软件定时器参数和对应任务信息，当涉及到定时任务的调整时，将调用etimer.c中的update\_time()，其中有如下代码段：

now = clock\_time();

t = timerlist;

/\* Must calculate distance to next time into account due to wraps \*/

tdist = t->timer.start + t->timer.interval - now;

for(t = t->next; t != NULL; t = t->next) {

if(t->timer.start + t->timer.interval - now < tdist) {

tdist = t->timer.start + t->timer.interval - now;

}

}

next\_expiration = now + tdist;

#if WITH\_DYNAMIC\_TIMEBASE

clock\_set\_expire(tdist);

#endif

该代码段计算所有定时任务中最先需要被执行的任务距离当前时刻的时间tdist，其中变量next\_expiration将保存该定时任务对应的时刻，动态时基机制增加了调用clock\_set\_expire (tdist)设置下一次中断的时间，以系统节拍数为单位。

在节点对应platform的clock.c中需要实现clock\_set\_expire() API，它需要根据输入的到时间隔节拍数设置MCU的硬件定时器在对应的时刻产生中断。在对应的中断处理函数中，系统要更新本地的时间计数值count和seconds，需注意的是每次增加的值应该是该次中断距上次所经过的节拍数，而不是像定时中断时每次增量都是1。

一般情况下，到时间隔较短，可以直接设置硬件定时器的一个比较值，当计数值达到该值时将产生中断。当到时间隔较长时，需要注意的是，硬件定时器（一般使用16位的）的计数值范围对应一个最大节拍间隔，如果设置的中断间隔大于该间隔，直接通过一次设置硬件定时器某个比较中断的比较值可能导致在错误的时刻执行任务；此外，当系统使用看门狗时，最大节拍间隔还需要小于看门狗的超时时间，否则节点将会重启。因此，较长的到时间隔需要分割为多个最大间隔和一个小间隔，也就对应多个定时中断。根据本系统的磁采样间隔和看门狗周期，采用最大间隔1s，每秒对应的节拍数是CLOCK\_SECOND=128，除去定时任务对应的必不可少的中断，每秒减少127次中断后MCU唤醒，将明显减少MCU能耗。

采用动态时基机制的节点工作电流情况如图 18所示，从图中可以看出，仅有约1秒左右才出现的一次约4~5mA的电流峰起，其余大部分时间都在休眠状态、电流小于20uA。

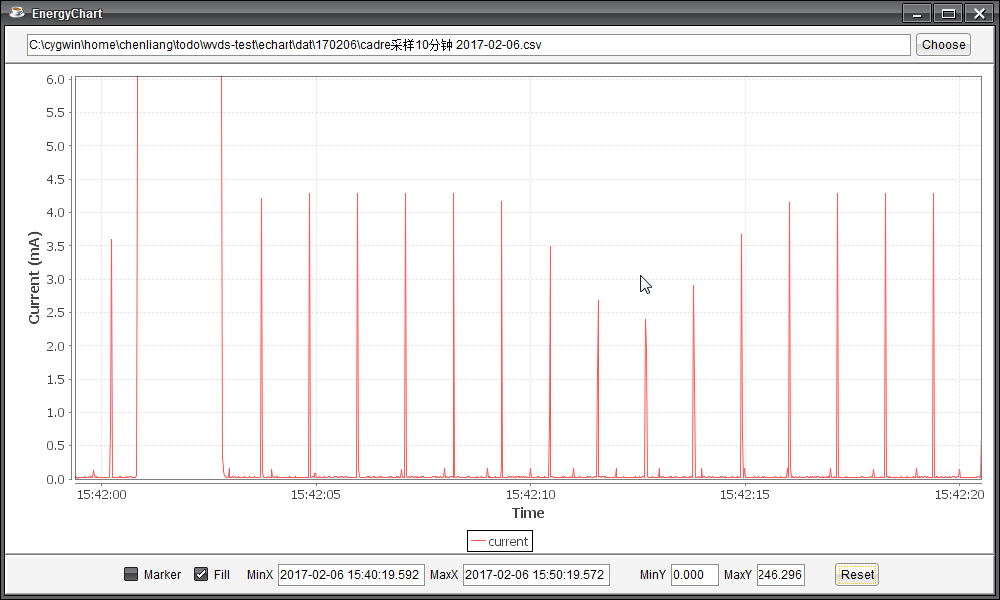


图 18 动态时基中断电流图

## 时间同步模块

本系统增加了contiki/core/sys/unixtime.c模块来维护节点本地的unix时间，并提供API可供其他模块调整本地时间实现与系统标准时间的同步。

根据是否具有硬件RTC模块，unixtime.c有两种实现方式，以编译标记UNIXTIME\_ARCH切换。定义UNIXTIME\_ARCH为1时，使用硬件RTC，对应的硬件平台代码需要提供unixtime\_arch\_get()、unixtime\_arch\_set()等API读取和控制硬件RTC；定义UNIXTIME\_ARCH为0时，采用软件实现模拟RTC。

unixtime.c实现为一个独立的process。采用软件实现时，unixtime\_process将启动一个etimer，每秒更新一次本地时间。使用time\_t类型的变量utime保存本地时间，在需要转换为6字节的BCD时间戳时，使用time.h中提供的API借助结构体struct tm进行转换得到对应的年/月/日/时/分/秒。为了减少unixtime\_process使用的单独etimer引入的冗余中断导致的功耗，已修改为依赖clock\_seconds()的实现。clock\_seconds()提供节点自启动后以秒计的本地时间，在etimer的中断中将会不断更新，配合utime作为本地unix时间偏移，即可得到完整的本地unix时间。在设置本地unix时间时，通过欲设置的时间减去clock\_seconds()即可得到正确的unix时间偏移。

本系统的AP/RP节点具有硬件RTC芯片DS3231，对应的驱动程序在dev/ds3231/ ds3231.c，同时使用platform/cadre1120/unixtime-arch.c作为unixtime.c的适配接口。

本系统的VD节点没有RTC芯片，因此采用软件实现。

为了实现节点的本地unix时间与中国标准时间保持一致，通过以下几个措施实现对节点本地时间的调整以及事件时间的修正：

1. AP在入网阶段发送时间同步消息0x05向服务器请求当前时间，根据接收到的响应消息中的时间戳调整本地的时间；
2. AP在接收到VD/RP的心跳消息时，在返回的响应消息中附加生成消息时的中国标准时间，VD/RP接收后根据该时间戳调整本地时间；
3. VD/RP在向AP发送的上行消息末尾附加4字节的节点本地unix时间值和时间戳同步标志，AP在接收时检查该标志、以及附加时间值和消息载荷中时间戳的差值，如果时间戳未同步，AP将对数据帧有效载荷中的时间戳按该时间差进行修正；
4. VD节点有时会在时间尚未同步的情况下保存泊位变化事件的数据，其时间不准确，在收到新的正确时间调整本地时间时，也会修正已保存事件的时间。

## 节点监测模块

节点监测模块的作用是让AP监测RP和VD的在线状态，基本思想是检查RP和VD是否正常地上报了心跳消息，在AP程序的主线程gateway.c中实现。

由于RP和VD可能采用不同的心跳周期，gateway中分别启动两个定时器周期性地检查RP和VD的上报情况。由于现场应用时节点的心跳周期较长，例如约30分钟，为了减少发现节点掉线的时间，检查周期定义为1个心跳周期加上一个短间隔如1分钟。根据前述心跳发送机制，每个节点在该时间段内最多会发送5~6次心跳消息，如果节点在正常运行，AP收不到该节点心跳消息的概率很小，因此未接收到就可以判定该节点掉线。

节点监测模块使用两个比特向量onlineBV和lastBV来记录节点的在线情况。onlineBV记录当前在线情况，如果接收到某个节点的心跳消息，则把onlineBV中该节点短地址对应的比特置为1；lastBV记录上一次检查时的在线情况，在周期性检查操作完成后将onlineBV复制为lastBV。

检查定时器到时后，调用app\_check\_offline()检查掉线情况，比较onlineBV和lastBV，如果某个节点对应的比特在lastBV中是1而在onlineBV中是0，则标记该节点掉线了，稍后发送上线掉线消息0x16向服务器报告。在检查完所有比特之后，将onlineBV复制为lastBV，并把onlineBV中当前为1的比特都清为0。

## 自检报警模块

自检报警模块的作用是检查节点各硬件子模块及电池电压/电量等是否正常工作，在出现故障时及时向系统报告以便进行相关的维护工作。模块的实现也分别在主线程文件device.c、router.c、gateway.c中。

自检报警模块在主线程使用一个专用etimer进行周期性的检查。当该etimer到时，调用app\_check\_health()检查各硬件子模块是否工作正常，发现工作异常时，比如电池电压低于对应的报警阈值，则置报警消息中对应的域的值为1；所有子模块检查完成后，如果存在一个或以上的故障，调用app\_send\_alarm()发送报警消息，否则不需要发送。

自检报警模块的检查周期由PIB.healthT决定，可以由系统在运行时进行调整，应当根据节点故障时所需的故障响应时间进行设置。

一个需注意的问题是，如果节点的射频出现故障，将无法上报报警消息，此时应该是通过发现节点长时间掉线而无法再自动上线发现该故障。

## 射频控制模块

如前所述，低功耗实现最重要的一点是对射频的控制，但各软件模块发送数据和等待响应的要求不同，需要对各模块的控制有一个协调，由射频控制模块实现，对应的实现代码在contiki\core\dev\radio.c中。

射频控制模块radio.c提供了以下几个API：

1. struct radio\_res \* radio\_alloc(char \*name)，申请一个射频锁，用自己的某指针存储；
2. void radio\_free(struct radio\_res \*rs)，释放自己持有的射频锁；
3. void radio\_on(struct radio\_res \*rs, clock\_time\_t timeout)，开启射频，并带有一个超时；
4. void radio\_off(struct radio\_res \*rs)，关闭射频。

通过以上几个API，需要对射频进行控制的模块，可以自行调用radio\_alloc()申请一个射频锁，在需要开启射频的地方调用radio\_on()，射频收发完成后调用radio\_off()关闭射频。

在有多个模块持有射频锁的情况下，只有所有射频锁都满足应该关闭射频时，射频才会被真正关闭，否则保持开启直到其他射频锁也正常或因超时而关闭射频。开启射频时，只有第一个射频锁开启时会执行开启射频操作，后续锁开启时无需真正操作。

radio\_on() API中的timeout参数指定了自己开启射频锁的最长时间，以系统节拍计。它的作用是防止模块自身因运行出错或未调用radio\_off()而导致节点一直不能关闭射频，该超时将有对应的ctimer软件定时器，在到时后将关闭该射频锁。调用radio\_on()时指定的timeout可以根据情况进行一些估计，比自己的射频任务完成时间较长一些没有关系，但较短可能导致射频提前被关闭而运行出错。

## 软件看门狗模块

本系统的各类型设备VD/RP/AP在实际部署后要再进行维护都相当麻烦，因此保证设备的稳定运行具有非常重要的意义，稳定性的实现非常依赖软件看门狗模块的作用，对应的实现代码主要是contiki\core\sys\taskmon.c和platform\mist-exp5438\watchdog.c。

platform\mist-exp5438\watchdog.c是节点平台对应的硬件看门狗控制模块，它提供对硬件看门狗停止、启动、喂狗等操作的支持，控制的是MCU上唯一的硬件看门狗。相比于原始版本的代码，增加了对taskmon模块的依赖，其中停止看门狗watchdog\_stop()和喂狗watchdog\_periodic()都依赖于taskmon\_ok()的返回值为1才真正进行操作，因此唯一的硬件看门狗逻辑上将可能扩展为多个软件看门狗来监测设备的运行。

contiki\core\sys\taskmon.c是软件看门狗的实现，taskmon是task monitor即任务监视器的缩写，它提供以下几个重要的API：

1. int task\_ok()，提供当前所有软件看门狗的状态，如果有一个的状态不正常，返回0表示不正常，所有都正常时返回1表示正常；
2. struct taskmon \*taskmon\_create(char \*name)，申请一个任务监视器；
3. void taskmon\_free(struct taskmon \*mon)，释放一个任务监视器；
4. void task\_begin(struct taskmon \*mon, clock\_time\_t timeout)，开始一个任务，带有超时；
5. void task\_end(struct taskmon \*mon)，结束一个任务。

每个任务监视器对应如下的一个数据结构：

struct taskmon {

char name[TASKMON\_NAME\_LEN]; // 任务名，便于调试时识别

uint32\_t beg; // 开始次数

uint32\_t end; // 结束次数

clock\_time\_t deadline; // 任务结束超时

uint8\_t running; // 本监视器是否在运行

uint8\_t pending; // 本监视器是否有任务正在执行中

};

软件看门狗模块支持的任务监视器的个数由TASKMON\_MAX\_NUM决定，默认值为8。

任务监视器的工作方式如下，对于一个关心的任务，为它申请一个任务监视器之后，在任务的开始位置，可以调用task\_begin()标记该任务已开始执行，同时指定了一个超时时间；如果任务正常运行结束，则在某位置应调用task\_end()标记任务已正常结束。task\_begin()会把beg计数加1，task\_end()会把end计数加1；如果在要进行喂硬件看门狗操作而调用taskmon\_ok()进行检查时，某个任务监视器的end和beg计数不一致，并且当前的时间已经超过了其deadline，说明该任务没有能够正常结束，这时taskmon\_ok()返回值为0就不会进行喂狗操作，超时后看门狗将导致节点重启。

以VD设备为例，保证其磁数据采样过程一直在正常运行是非常重要的，采用了两个任务监视器分别名为sample和sensor来监视其过程。sample监视采样的周期性是否正常，第一次启动采样定时器时也调用task\_begin(sample,...)，在采样定时器到时，向磁传感器发出采样指令时，调用task\_end(sample)结束第一次周期性检查，又立即调用task\_begin(sample,...)监视下一次采样执行。sensor监视传感器的采样是否正常返回数据，在向磁传感器发出采样指令的位置调用task\_begin(sensor,...)，在磁传感器正常完成了采样的回调函数中调用task\_end(sensor)，如果磁传感器未能正常返回采样结果，则sensor失败导致重启。

调用task\_begin()时指定的超时时间可以是多次任务正常执行的时间，从而可以容忍几次任务执行失败。还是以上述sensor为例，假设节点正常每1秒采样一次，则调用task\_begin(sensor, (CLOCK\_SECOND \* 4))时，如果第1次采样因出错未能返回数据，sensor监视器因为还没超时仍然认为是正常，程序还可能正常运行进行第2次采样；如果第2、3次采样都失败，sensor监视器才会因超时判定出错而重启，只要有一次成功，则会调用task\_end()结束本次任务执行，然后在下次开始采样时又重新计超时。

## 启动引导模块

启动引导模块又可称为引导程序，它作为一个独立的程序在节点上电后首先运行，然后根据某些内部存储器中的数据决定是根据串口接收数据或外存储器中数据更新程序存储器中的数据，还是直接运行合法的主应用程序。本系统引导程序支持从串口或外存储器更新设备的主程序。

引导程序本身占据MCU的程序存储器的0xF000-0xFFFF区间，主程序则占据0x5C00-0xEFFF和0x10000-0x1FFFF区间。引导程序和主程序在编译时需要使用修改过的链接文件apps\wvds\{boot,device,router,gateway}\EW430\config\linker\lnk430f5438a.xcl，而不是默认的链接文件，这样才能使得程序内容被安排在前述的地址区间内。

引导程序不能使用任何中断，各中断处理函数都像如下示例代码，它的作用是直接跳转到主程序对应的中断处理函数处进行后续处理，相当于把原本0xFF80-0xFFFF的中断向量整体移动到0xEF80-0xEFFF，从而各中断都能由主应用程序的相应函数进行处理。

ISR(UNMI, unmi\_isr)

{

asm(" br &0xEFFA;");

}

ISR(SYSNMI, sysnmi\_isr)

{

asm(" br &0xEFFC;");

}

引导程序将相关的数据保存在MSP430 MCU的内部信息存储器中，起始地址是0x1800。涉及到的相关数据定义在结构体struct BOOT中：

struct BOOT {

uint16\_t magic; // 初始化标志

uint8\_t role; // 节点类型. 1:VD, 2:RP, 4:AP

uint8\_t version; // 固件版本

uint8\_t update; // 更新来源. 0:无需更新, 1:串口, 2:外部Flash

uint8\_t flash; // Flash编号. 0:VD, 1:RP/AP

uint32\_t size; // 新固件大小

uint32\_t addr; // 新固件起始地址

uint16\_t crc; // 新固件校验和

uint8\_t updver; // 新固件版本

};

其中，BOOT.role记录了节点的设备类型，可以增加检查以防止错误的固件用于某设备；BOOT.version记录了节点固件的版本号；BOOT.update标记了运行模式，如果值为0，表示无需进行更新，直接运行现有的应用主程序，如果为1，表示等待从串口接收新的主程序进行更新，如果为2，表示将从外部Flash读取新的主程序并进行更新。BOOT.flash/size/addr/crc/ updver则记录了存储在外部Flash中的新主程序的相关信息。

引导程序在串口更新模式时，接收更新数据0x18消息有效载荷除时间戳部分（对应如下结构体app\_data\_ota\_data的定义）的封装数据帧，每接收一帧，将根据当中的地址直接写入MCU程序存储器。数据帧的封装采用类似RFC1622 HDLC的简单方法，即以0x7E字节起始和结尾，有效数据部分每出现一个0x7E或0x7D字节，发送端分别转义替换为0x7D 0x5E和0x7D 0x5D两个字节，接收端则相应逆转义得到原始数据字节。经过该封装的数据帧在串行通信出现错误时能够正确找到下一帧数据的起始字节，从而在经过重传时能实现所有固件数据通过串行通信发送到MCU。

struct app\_data\_ota\_data {

uint8\_t type; // 目标节点类型: VD,RP,AP

uint8\_t ver; // 本段数据所属固件的版本号

uint8\_t beg:1; // 第一个数据包应为1，其它应为0

uint8\_t reserva:6;

uint8\_t fin:1; // 最后一个数据包应为1，其它应为0

uint8\_t reservb;

uint8\_t addr[4]; // 本段数据在程序存储器中的起始地址

uint8\_t len[2]; // 本段数据的长度，总为128

uint8\_t data[128];

uint8\_t crc[2]; // 从type域到data域的CRC16校验和

};

引导程序在Flash更新模式时，从外部Flash中读取新的固件数据，以程序存储器的页大小（512字节）为单位逐页从外部Flash读取数据并写入MCU程序存储器。新的固件数据在通过无线更新模块接收后保存在外部Flash中，起始地址为BOOT.addr，数据大小是BOOT.size，在有多于一个外部Flash的情况下，BOOT.flash为数据在哪个Flash中。

需要注意的是，新固件的主程序数据在外部Flash中仅按实际内容连续存储，未保存在程序存储器中对应地址等信息；如前所述，主程序在程序存储器中的地址区间是0x5C00-0xEFFF和0x10000-0x1FFFF，因此在读取外部数据将0x5C00-0xEFFF区间覆写完毕后，后续数据需要跳转到0x10000地址继续覆写。

## 无线更新模块

无线更新模块的功能是将新的固件文件发送到网络中的每个节点上并存储到存储器，然后直接或调用其他模块完成用新固件更改MCU程序存储器中数据实现设备功能的更新。无线更新模块的实现在apps\wvds\deluge.c中，它是基于官方的实现代码、根据生产环境的使用需求进行了修改。

### 固件数据解析

如前所述，新版本程序按照Release配置编译后将生成只包括主程序内容的文件如device.txt和与引导程序合并的文件如device-boot.txt，进行无线更新时只能使用device.txt文件。要把该文件包含的程序数据发送到目标更新节点上，首先要解析该文件构造向节点发送的数据包集，对应的实现在文件apps\wvds\t\gui\src\wvdsota\src\main\java\net\tinyos\ demo\DlgOTAUpdater.java的downloadFirmware()方法中。

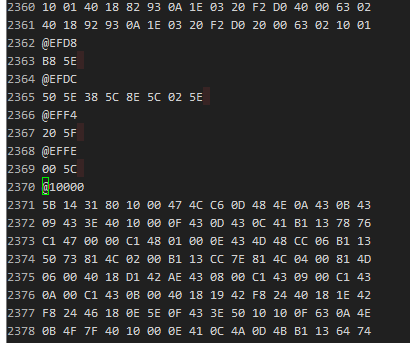


图 19 固件文件片段

受本系统数据帧定义的长度限制，完整的固件数据需划分为每128字节一个数据包。固件文件为msp430-txt格式的文本文件，如图 19所示，绝大多数行是16字节数据对应ASCII字符串的数据行，少量行是以@开头的地址行，表示后续行数据从该地址开始。对数据行，需要转换为16字节数据，比如图 19中2371行转换后为{0x5B,0x14,0x31,0x80,0x10,0x00,0x47, 0x4C,0xC6,0x0D,0x48,0x4E,0x0A,0x43,0x0B,0x43}，添加到当前数据包的尾部，并更新已添加字节的对应地址；当前数据包的总长度达到128时，重新开始构造下一个数据包。对于地址行，需要检查已添加的最后一个字节数据对应地址和当前地址之间是否有空缺，如果有空缺，需要以字节0xFF填充，比如图 19中2066至2069行，0x20 0x5F在地址0xEFF4，0x00 0x5C在地址0xEFFE，中间地址0xEFF6至0xEFFD没有对应数据，在构造的数据包应该填充对应数量的0xFF字节。

### 固件数据分发

固件数据分发过程就是将解析阶段构造的数据包集发送到目标更新节点上的过程。

分发的第一阶段是上位机将固件数据通过串行通信发送到更新节点（烧录apps\wvds\ updater程序的VD节点）上，每次发送一个WVDS数据帧，其中有效载荷的定义见如下结构体struct app\_data\_ota\_data，包括128字节的有效固件数据。更新节点接收数据后保存到外部Flash中，对不同设备的固件文件，将保存到Flash中的不同区间。第一阶段的分发由配套的基于Java上位机软件WVDS-OTA发起和发送数据，更新节点完成数据接收和保存到外部Flash。WVDS-OTA软件的使用方法可参见其帮助文档《WVDS-OTA使用手册》，它对应的软件源代码在apps\wvds\t\gui\src\wvdsota目录。

#define APP\_OTA\_DATALEN 128

struct app\_data\_ota\_data {

uint8\_t tstamp[TSTAMP\_LEN];

uint8\_t type; // 目标节点类型: VD,RP,AP

uint8\_t ver; // 本段数据所属固件的版本号

uint8\_t beg:1; // 第一个数据包应为1，其它应为0

uint8\_t reserva:6;

uint8\_t fin:1; // 最后一个数据包应为1，其它应为0

uint8\_t reservb;

uint8\_t addr[4]; // 本段数据的起始地址

uint8\_t len[2]; // 本段数据的长度，总为APP\_OTA\_DATALEN

uint8\_t data[APP\_OTA\_DATALEN];

uint8\_t crc[2]; // 从type域到data域的CRC16校验和

};

更新节点使用CFS API来访问外部Flash进行固件文件的读写，对应的实现代码在apps\wvds\cfs-flash.c中，通过自有代码实现对多文件的支持而又不采用coffee文件系统的复杂实现，不采用coffee文件系统的一个重要原因是避免引导程序从外部Flash读取固件数据所需的实现代码太复杂。由于要支持VD/RP/AP多种设备的升级，更新节点需要支持3个对应文件的读写。

分发的第二阶段是更新节点通过无线通信将固件数据发送给目标节点。更新节点从外部Flash读取数据，按照deluge协议的设计，整个固件数据以page为单位进行传输，每个page包含256字节，受无线通信数据包长度限制，每个数据包只能包含64字节有效数据，因此每个page又划分为4个数据包。

deluge协议使用4种类型的消息完成更新文件数据的分发：

* summary消息，告知邻居节点自己已持有的文件版本号和page数目；
* profile消息，告知邻居节点新版本文件的版本号、page总数等信息；
* request消息，向更新源节点请求特定版本、特定page的全部或部分packet数据包；
* packet消息，更新源节点向请求节点发送、包含有效的固件数据。

deluge协议简单来说，各节点发送summary消息，源节点接收到某节点的summary消息发现其版本较低时，向其发送profile消息告知有新版本数据；接收到profile消息的节点进入RX状态，向源节点发送request消息；源节点接收到某节点的request消息时，根据其请求的page序号和数据包掩码发送packet消息；接收packet消息的节点在完成当前page所有数据包的接收后将发送request消息请求下一个page的数据，否则请求当前page中未接收到的packet数据包；被更新节点和源节点间重复发送request和发送packet过程直到被更新节点接收到新文件数据的所有page。按照上述机制，deluge协议适合网络中源节点将文件数据分发到网络中的每个节点。

原始版本的deluge协议实现存在一些不适合现场应用需求的情况：1、不能分发多个设备对应的固件文件，每个设备由于功能不同需要使用不同的固件文件；2、启动分发后必定是网络中所有节点都参与分发并接收到数据，不适合逐步升级时只希望部分节点接收数据以及希望节点不接收不适用于自己的数据的情况；3、不适应VD需要一直保持射频关闭但是又能进行无线更新的要求；4、不能满足更新超过64kB的大文件且要占用过多的RAM。

修改版本的deluge协议实现的一个重要改动是在各消息的定义中增加了mode和target域，类似如下的packet消息定义：

struct deluge\_msg\_packet {

uint8\_t cmd;

uint8\_t mode;

uint8\_t target[8];

deluge\_object\_id\_t object\_id;

uint8\_t version;

uint8\_t pagenum[2];

uint8\_t packetnum;

uint8\_t crc[2];

unsigned char payload[S\_PKT];

};

其中mode表示更新模式，target表示更新的目标节点。

更新模式定义了三种：1、单个节点，只更新单个目标节点；2、一跳所有，只更新源节点所有的一跳邻居节点；3、整个网络，更新网络的所有节点。在单个节点的更新模式下，target域设置为目标节点的MAC地址，接收到deluge消息时检查该域和自身MAC是否一致，只有一致的节点才会响应，从而实现只更新单个节点。单个节点和一跳所有的更新模式使得灰度升级成为可能，即先更新部分节点验证新版本的功能，之后再进行整个网络的更新。目前仅限于使用单个节点更新模式，更新网络需要逐个节点完成更新。

不同的更新模式对协议中的一些时间参数有不同的要求，比如在整个网络或一跳所有更新模式下，节点接收一个page后需要在较长的随机延时后（避免与其它节点的请求冲突）再请求下一个page，而已有更多page的发送节点应该等待较长时间收到多个请求后再发送page的有效数据，而在单个节点更新模式下，以上两个时间均可减少，以便缩短总体数据传输的时间。

为了满足更新超过64kB大文件的要求，原消息中uint8\_t类型的pagenum被修改为uint16\_t类型，因此可分发的数据大小超过256\*256=64kB，最大可达256\*65536=16384kB。原实现中deluge\_object有struct deluge\_page pages[]域，存储了更新文件所有page的元数据，在page数目很多时将占用大量RAM，修改版本则更改为只存储当前传输page的元数据。

更新协议的另一项修改是将节点的更新进度，即已接收的page数目保存到镜像文件元数据并写到外部Flash中，在节点意外重启的情况下，通过检查进度可以自动继续之前尚未完成的更新过程。该修改提供了类似断点续传的支持，可以在意外情况下尽快完成某节点的更新过程。

# 多任务交互

节点程序在运行时，主线程作为AUTOSTART线程由main入口函数在初始化后启动，参见节点平台主程序文件contiki-exp5438-main.c中的代码行：

autostart\_start(autostart\_processes);

之后，主线程可能通过多种方式启动相关的子线程，这些子线程分别处于其专注的某种事件和相关数据，再通过多种方式通知其他线程与之配合进行相关处理。

以VD节点为例，节点正常运行时将有如下列表中的线程在运行：

|  |  |
| --- | --- |
| 线程名称 | 功能 |
| device\_process | VD程序主线程 |
| cc11xx\_process | 射频驱动程序线程 |
| sensors\_process | 传感器线程 |
| etimer\_process | 事件定时器线程 |
| ctimer\_process | 回调定时器线程 |
| netcmd\_process | 下行命令线程，负责缓存和转发面向休眠节点的下行命令 |
| logger\_process | 运行日志线程，负责射频发送错误/警告/调试消息供sniffer诊断 |

表 11 VD运行线程列表

线程的启动有以下几种方式：

1. 作为系统线程，由主程序启动，如上表中的etimer\_process、ctimer\_process和sensors\_process；
2. 线程模块本身提供API供调用者启动该模块，在该API中负责启动自己的线程，例如调用NETSTACK\_RADIO.init()将实际调用cc11xx.c中的init()子函数，该函数有代码行

process\_start(&cc11xx\_process, NULL);

实现cc11xx\_process线程的启动。这类有cc11xx\_process、netcmd\_process、logger\_process；

1. 主线程直接调用如process\_start(&foo\_process, NULL)启动某子线程。

线程间的互动也存在几种不同的实现方式：

1. 通过回调函数。以数据无线接收过程为例，主线程启动时调用mesh\_open()打开了一个mesh通信连接，指定了一个函数app\_mesh\_recv()作为接收到数据包时的回调函数。在有属于该mesh连接的数据发送到某节点时，cc11xx\_process通过处理接收中断首先获得该数据包，然后通过NETSTACK\_RDC.input()交由上层处理，经过多次函数调用，最后到达mesh.c模块，它将通过mesh\_open()时注册的回调函数调用app\_mesh\_recv()交给主线程如gateway\_process处理；
2. 通过产生一个process\_event由系统调度器调度。继续第1步的例子，在app\_mesh\_recv()中，新接收的数据包将从packetbuf复制到接收消息队列中，AP节点程序gateway.c中调用如下代码行：

process\_post(&gateway\_process, mesh\_rcvd\_event, NULL);

产生一个mesh\_rcvd\_event，通过该调用，系统线程调度器将记录gateway\_process一个 该事件要处理，并在之后通过call\_process()调用gateway\_process进行处理。系统线程 etimer\_process也采用这种方式；

1. 通过产生poll事件由系统调度器调度。这也是一种很常用的方式，在把数据存放到类似packetbuf模块或全局变量之后，可以通过类似代码行 process\_poll(&foo\_process) 让该线程稍后处理。

# 软件仿真和测试用例及测试方法

## Cooja软件仿真

较简单的方式是使用InstantContiki环境进行仿真，基本的使用方法可以参见在线文档[https://mintisan.github.io/notes/get-started-with- contiki.html](https://mintisan.github.io/notes/get-started-with-contiki.html)。该虚拟机安装了仿真所需的完整工具链，特别适合于编译为嵌入式平台对应的虚拟节点进行仿真。

建立仿真时，可以选择两种虚拟节点：一是Trxeb1120虚拟节点，仿真器将调用msp430 gcc编译器完成编译，程序在虚拟的MSP430 MCU上运行仿真。它的优点是编译出的程序与最终在硬件节点上可运行的基本一致，缺点是受RAM限制无法打印出足够多的日志方便进行协议栈的分析；二是Cooja虚拟节点，仿真器将调用主机平台的gcc编译器完成编译，作为主机上可运行的程序与仿真器Java部分交互进行仿真。优点是可以打印出较多的日志方便进行协议分析，缺点是与硬件节点上使用的程序在底层有所不同。

## 仿真分析方法

在运行仿真后，主要的分析方法是对输出的日志文件过滤出特定关键字的行，查看预期的事件是否已经发生，比如预期的数据包是否已接收到。

对于程序运行时可能出错的位置，日志行中建议添加如warn、error的关键字，针对日志文件过滤这些关键字可以快速找出可能需要修正的错误。

在代码中添加调试日志行时，输出内容建议以该软件模块的完整名或简称开始，则分析日志可以过滤该关键字着重分析该模块的运行情况。

对于一些协议栈组件如Rime栈中mesh模块、trickle模块、无线更新deluge模块，可以编写单独的小测试程序进行仿真，主要是熟悉其在网络中的运行机制和实际表现，可以学习官方代码中的一些示例构建仿真。

对本系统级别的完整程序进行软件仿真是一个相当复杂的问题，其中涉及到对诸如磁传感器、GPRS模块、后台服务器等增加一个简单的软件仿真实现，需要对程序主体代码根据仿真情况进行相当多的修改，要求开发人员有很强的能力根据情况添加适应仿真环境的简化修改。本系统程序开发后期也未再进行软件仿真，因此对完整程序的仿真不再深究。软件仿真的一些情况也难以符合实际节点上的运行，也是不值得花费太多时间的原因。

# 附录一

Cygwin环境一键安装脚本

1. 从http://www.cygwin.com下载setup-x86.exe或setup-x86\_64.exe，存放到一个新建的软件目录下，比如D:\software\cygwin；
2. 在第1步的新建目录下创建文件install.bat，输入以下内容：

@echo off

set PATH=C:\Windows\System32;C:\Windows\SysWOW64

set CYGROOT=C:\cygwin

set CYGSITE=http://mirrors.ustc.edu.cn/cygwin

set CYGDIR=%~dp0

set PACKAGES=vim,git,python2

setup-x86.exe --quiet-mode --no-verify --root %CYGROOT% --site %CYGSITE% --local-package-dir %CYGDIR% --arch x86 --packages %PACKAGES%

3. 双击install.bat则自动从中国科技大学的cygwin服务器镜像下载并安装指定的多个软件包，即脚本中的PACKAGES变量。