**协议栈大作业**

**按需移动自组网络？？？**

学 院（系）： 软件学院

专 业： 网络工程

学 生 姓 名： 李昌妮 王馨怡 卢琦

学 号：

指 导 教 师：

评 阅 教 师：

完 成 日 期：

大连理工大学

Dalian University of Technology

目录

[1 概述 3](#_Toc28866590)

[1.1 特点 3](#_Toc28866591)

[２　路由消息介绍 4](#_Toc28866592)

[RREQ 4](#_Toc28866593)

[RREP 4](#_Toc28866594)

[RRER路由错误 4](#_Toc28866595)

[RREP－ACK路由回复确认 4](#_Toc28866596)

[３　路由过程 6](#_Toc28866597)

[３.１运行流程 6](#_Toc28866598)

[３.２路由过程简述 7](#_Toc28866599)

[４　代码介绍 8](#_Toc28866600)

[4.1　源文件 8](#_Toc28866601)

[4.2 代码分析 8](#_Toc28866606)

[4.3.2 其他源文件 15](#_Toc28866607)

[2.2.2 Aodv\_timeout.c分析 22](#_Toc28866608)

[2.2.4 Aodv\_socket.c分析 26](#_Toc28866609)

[2.2.6 Locality.c分析 28](#_Toc28866610)

[2.2.7 aodv\_rrep.c 28](#_Toc28866611)

[2.2.8 aodv\_neighbor.c 35](#_Toc28866612)

[2.2.9 aodv\_hello.c 38](#_Toc28866613)

[2.2.11 routing\_table.c 41](#_Toc28866614)

[2.2.13 aodv\_rerr.c 50](#_Toc28866615)

[2.2.14 nl.c 53](#_Toc28866616)

[2.2.15 defh.h 59](#_Toc28866617)

[2.2.16 endian.c 59](#_Toc28866618)

# 概述

Aodv是无线自组网按需平面距离向量路由协议（Ad hoc On-Demand Distance Vector Routing）。是应用于无线随意网络中进行路由选择的路由协议，它能实现单播和多播路由。该协议是Ad Hoc网络中按需生成路由方式的典型协议。在网络资源充分的情况下，AODV协议可以通过定期广播hello报文来维护路由，一旦发现某一个链路断开，节点就发送ERROR报文通知那些因链路断开而不可达的节点删除相应的记录或者对已存在的路由进行修复。aodv能在动态变化的点对点网络中确定一条到目的地的路由，并且具有接入速度快，计算量小，内存占用低，网络负荷轻等特点。aodv采用目的序列号来确保在任何时候都不会出现回环，避免了传统的距离向量协议中会出现的很多问题。由于Aodv是属于网络层的协议，所以每次寻找路由时都要触发应用层协议。

## 特点

(1)AODV属于网络层协议。每次寻找路由时都要触发应用层协议，增加了实现的复杂度。[2]

(2)IEEE 802.15．4规定的帧大小为127字节，MAC头部及尾部校验最多使用了25字节，IPv6头部与UDP头部占用48字节，只剩下54字节的空余。

(3)LR—WPAN中，拓扑结构相对简单，网络的规模相对较小，节点的位置不固定，对它的设计首先要考虑的因素是简单、节能等问题。AODV的路由框架和报文设计并没有考虑这些问题。

无线[自组网](https://baike.baidu.com/item/%E8%87%AA%E7%BB%84%E7%BD%91" \t "_blank)按需平面距离[矢量路由协议](https://baike.baidu.com/item/%E7%9F%A2%E9%87%8F%E8%B7%AF%E7%94%B1%E5%8D%8F%E8%AE%AE" \t "_blank)（Ad hoc On-Demand Distance Vector Routing，AODV），是应用于无线[网状网络](https://baike.baidu.com/item/%E7%BD%91%E7%8A%B6%E7%BD%91%E7%BB%9C" \t "_blank)（也称作[无线mesh网络](https://baike.baidu.com/item/%E6%97%A0%E7%BA%BFmesh%E7%BD%91%E7%BB%9C" \t "_blank)）中进行路由选择的路由协议，它能够实现[单播](https://baike.baidu.com/item/%E5%8D%95%E6%92%AD" \t "_blank)和[多播](https://baike.baidu.com/item/%E5%A4%9A%E6%92%AD" \t "_blank)路由。该协议是Ad Hoc网络中按需生成路由方式的典型协议。

# ２　路由消息介绍

## RREQ

## RREP路由回复

目的节点或可通向目的节点的中间节点收到RREQ消息后单播RREP消息：

8b 1b 1b 9b 5b 8b

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type  类型 | R | A | Reserved  保留字段，填充0 | Prefix Size  前缀长度 | Hop Count  跳数 |
| Destination IP Address  目的节点IP地址 | | | | | |
| Destination Sequence Number  目的节点序列号 | | | | | |
| Originator IP Address  源节点IP地址 | | | | | |
| Lifetime  生命周期 | | | | | |

## RERR路由错误

连接断开而导致有节点不可达时，发出RERR消息

1b

8b

15b

8b

|  |  |  |  |
| --- | --- | --- | --- |
| Type  种类（3） | N | Reserved  保留字段，填充0 | DestCount  不可达节点数目 |
| Unreachable Destination IP Address  不可达目的节点的IP地址 | | | |
| Unreachable Destination Sequence Number  不可达目的节点的序列号 | | | |
| Additional Unreachable Destination IP Address (if needed) | | | |
| Additional Unreachable Destination Sequence Number (if needed) | | | |

## RREP－ACK路由回复确认

当收到’A’被置位的RREP消息，必须回发路由回复确认消息

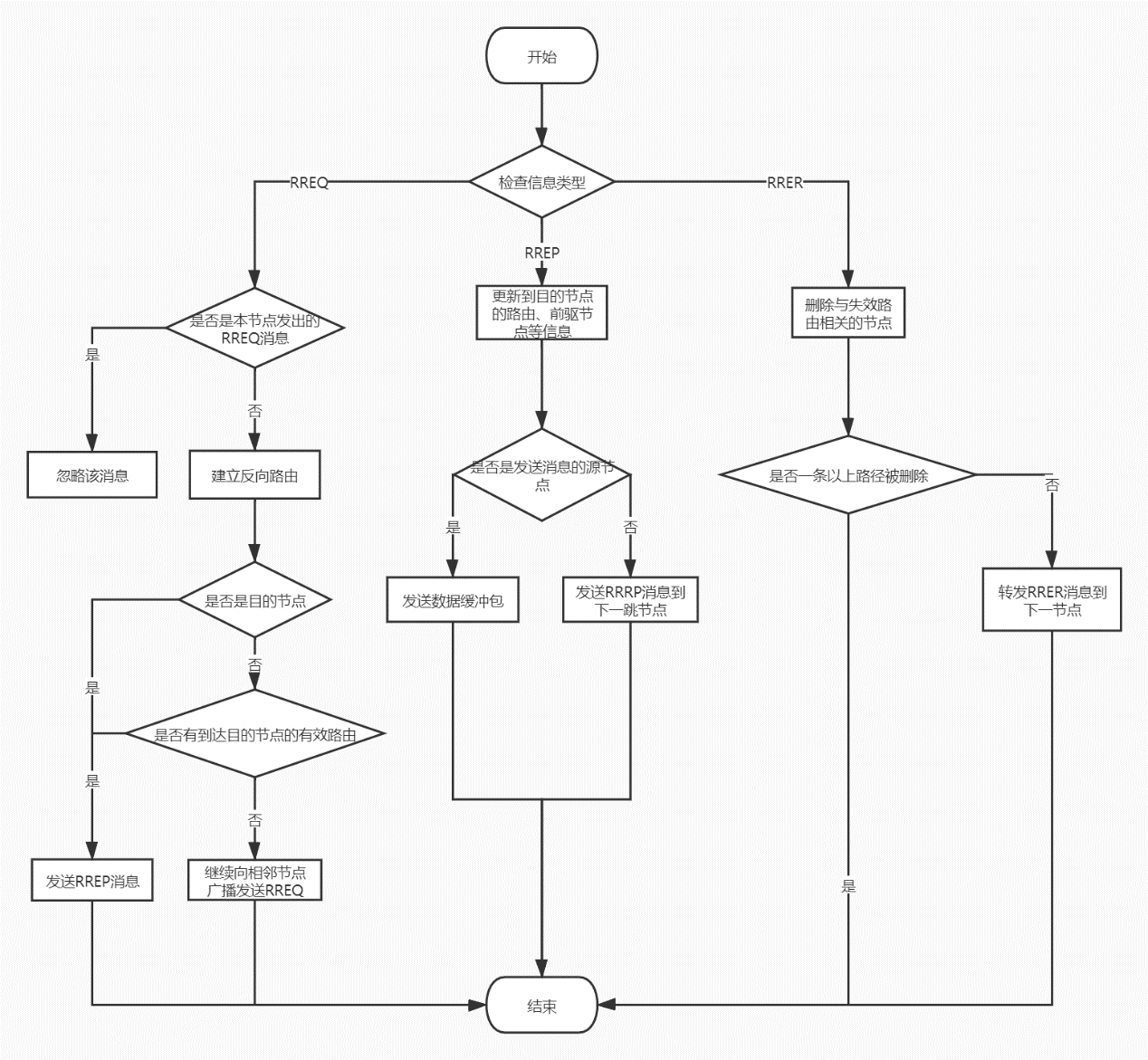
8b

8b

|  |  |
| --- | --- |
| Type  种类（4） | Reserved  保留字段，填充0 |

# ３　路由过程

## ３.１运行流程



先驱表为空

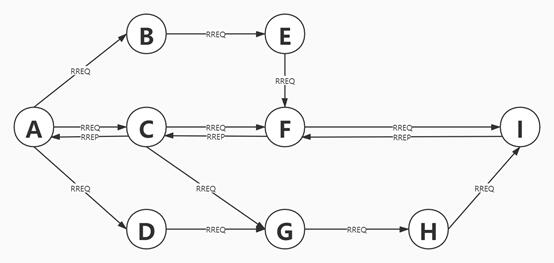
RREP

流程图从一个节点接收到一条aodv消息开始。

首先检查消息类型

* 如果是RREQ消息
* 判断自己是否需要对其进行处理
* 建立反向路由，更新RREQ
* 判断自己是目的节点还是中间节点
  + 是目的节点则根据反向路由单播RREP节点
  + 是中间节点则判断是否有到目的节点的路由
  + 是中间节点且没有到目的节点的路由，继续广播更新后的RREQ
* 如果是RREP消息
  + 更新到目的节点的信息
  + 判断自己是否为源节点
    - 是则路由建立完成，可以发送数据包
    - 否则继续转发更新后的RREP消息
* 如果是RRER消息
  + 删除失效路由
  + 判断自己是否先驱表为空
    - 是则结束
    - 否则新建RERR消息并转发给邻居节点

## ３.２路由过程简述



假设节点A要发送数据包到节点I，则节点A要与节点I建立连接。

* 路由请求
* A向周围节点广播RREQ消息。
* 邻居节点B、C、Ｄ判断接收到的RREQ消息是否为自身节点发出的RREQ消息，并判断自己是否是目的节点，若不是，则继续向相邻节点广播RREQ消息，并建立反向路由。
* 路由回复
* 直到某个节点发现自己是目的节点，则打包一些信息生成RREP消息并将其按照反向路由发送。在发送信号之前它会先更新自己的序列号，将其增１。
* 当消息发送到下一跳节点时，更新跳数等信息，创建转发路由，并判断是否为源节点。如果是则路由建立完成，不是则继续转发，直到路由建立完成。
* 路由检测
* 当建立好链路之后，节点之间通过Hello消息确定链路状态是否正常。
* 如果一定时间内未收到某相邻节点的Hello消息，则视作该路径失效，并发起局部修复。
* 局部修复失败则会发送RERR消息进行无效路由的删除。

# ４　代码介绍

## 4.1　源文件

AODV的所有代码包含在十七个C文件和17个头文件中，在子目录下还存在两个文件夹——lnx和patches，其中lnx中包含了AODV的核心代码。该协议的主要C文件以及它们的作用列于图2-1中。

|  |  |
| --- | --- |
| 文件 | 说明 |
| Aodv\_hello.c | 添加活跃邻居节点以及处理邻居节点断开 |
| Aodv\_socket.c | 管理AODV套接字 |
| Aodv\_timeout.c | 超时处理 |
| Debug.c | 记录日志 |
| Endian.c | 补充缺乏<endian.c>的系统 |
| List.c | 链表操作 |
| Llf.c | 链路层反馈 |
| Locality.c | 寻找目的地的方位 |
| Nl.c | AODV协议专用的套接字 |
| Main.c | 协议的初始化和组织运行 |
| Routing\_table.c | 路由表 |
| Seek\_list.c | RREQ正要寻找的目的链表 |
| Timer\_queue.c | 定时器有关操作 |
| Adov\_rreq.c | RREQ消息 |
| Aodv\_rrep.c | RREP消息 |

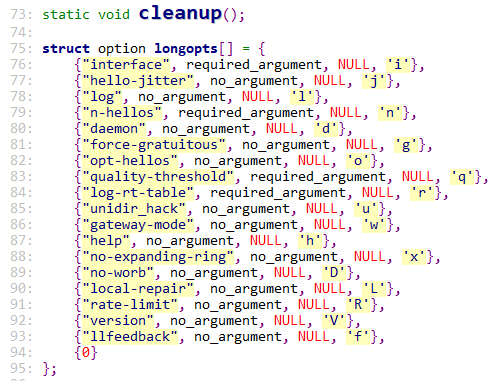
图2-1 主要C文件



## 代码分析

#### main.c

|  |  |  |
| --- | --- | --- |
| 函数名 | 返回值 | 功能 |
| usage(int status) | Void | 如果状态值为0则输出usage信息，不为0打印标准错误提示用户进行帮助查看 |
| set\_kernel\_options() | int | 把ip\_forward写入on，置为开启。把所有接口关闭接口下发送和接受重定向的文件写入终止符off，在所有接口禁用ICMP重定向 |
| find\_default\_gw(void) | int | 寻找默认网关 |
| get\_if\_info(char \*ifname, int type) | struct sockaddr\_in \* | 获取指向名为ifname的接口地址的指针，如果设备驱动失败返回空指针 |
| attach\_callback\_func  (int fd, callback\_func\_t func) | int | 增加一个文件描述符为fd，功能函数为func的回调结构元素，同时限制当前所拥有的套接字的处理函数和文件描述符数量不超过限制 |
| load\_modules(char \*ifname) | void | 根据接口的名字来获取其对应的设备信息以加载模块，没有则调用modprobe来加载模块。加载完成后还会检查是否加载完成。 |
| remove\_modules(void) | void | 调用system函数卸载模块 |
| host\_init(char \*ifname) | void | 主机初始化。如果没有接口，则先寻找无线接口。之后将rreq\_id初始化为0，已启用接口数为0，查找要在其上广播的所有接口并存储其信息，加载内核模块，启用IP转发并设置其他内核选项。 |
| signal\_handler(int type) | void | 信号处理器，使用switch结构根据信号的种类做不同处理。如果是SIGSEGV则写一条日志，其他信号不做处理。 |
| cleanup(void) | void | 静态函数，用于清空路由表、套接字、关闭日志、关闭套接字、卸载模块等。 |



* 静态函数，指明了一个结构体option类型的长参数longopts的名称和性质。

76~94 {name,has\_arg,flag,val}

name:长参数名

has\_arg:为required argument表示该参数后一定要跟个参数值，为 no\_argument表示该参数后不跟参数值

flag:为NULL表示getopt\_long()返回最后一个参数val

val:发现长选项时的返回值



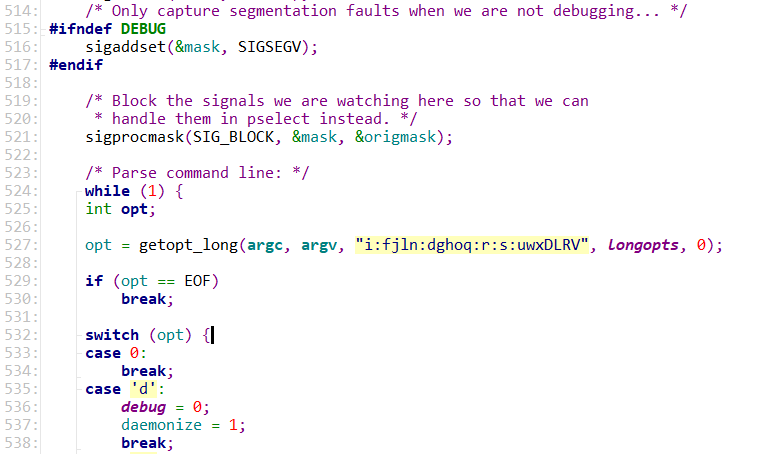
482~489 声明一些变量

492~498 设置程序名为其地址

501 打开调试输出开关

507~509 服务器关闭一些信号

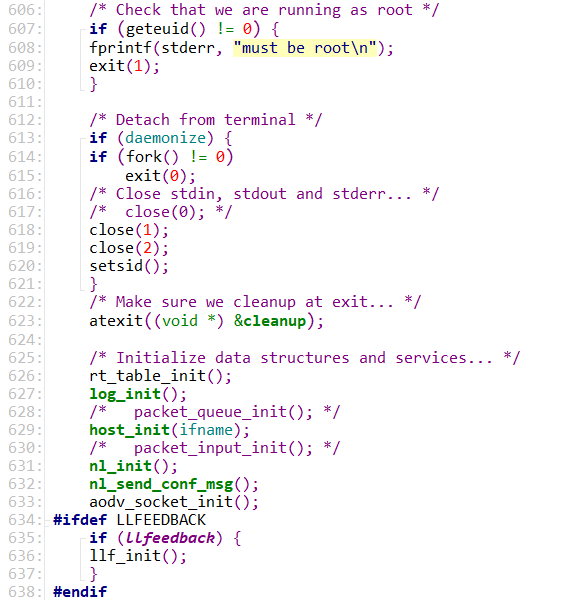
511~513 把这些信号放进屏蔽信号集里



515~517 如果调试开启则屏蔽SIGSEGV信号

521 阻塞正在观看的信号之后可以使用pselect来处理

523~… 处理命令行操作，使用循环结构，根据用户输入进行不同操作，其各个输入参数在usage函数中有定义。

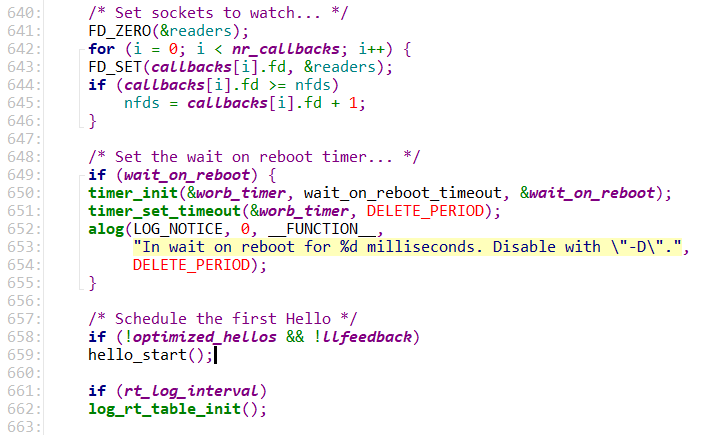


607~610 检查确定自己是在root权限下

612~621 如果daemonize 变量不为0，则关闭标准输入输出流和标准错误流，当前进程如果是父进程则直接退出，如果是子进程则并且使当前进程成为新的会话的领头进程，并与其父进程的会话组和进程组脱离。由于会话对控制终端的独占性，进程同时与控制终端脱离。

623 atexit函数是正常程序退出时调用的函数，我们把他叫为登记函数

626~637 各个数据结构和服务的初始化



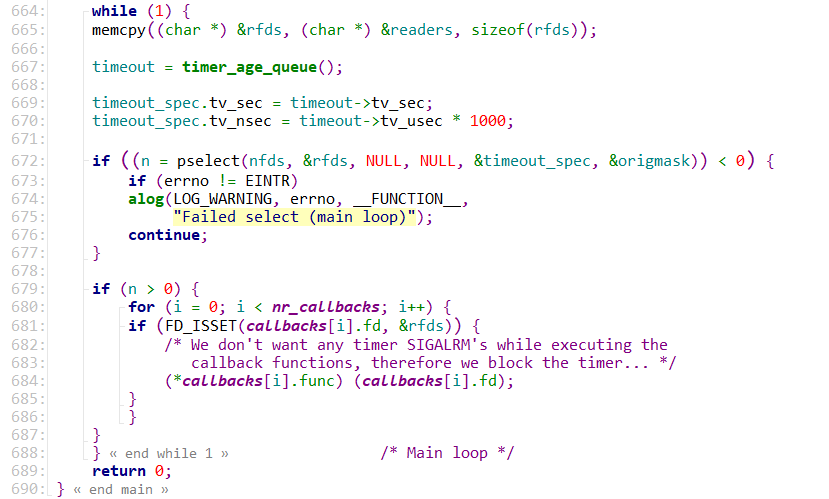
641 将指定的文件描述符清空

642~646 添加进文件描述符集合，更新文件描述符数量

648~655 设置超时重启定时器并写日志

657~659 准备第一个hello帧

661~662 初始化路由表



* 一个while永真循环

665 把文件描述符readers拷贝到rfds指向的数组中。

667~670 设置超时定时器

672~677 使用pselect函数进行IO复用，监视多个文件描述符的集合，用超时轮询的方式查看文件的读写错误可操作性，判断是否有符合条件的事件发生。

679~687 如果有一个或多个事件发生则查看其文件描述符如果在集合里则调用相应的回调函数来处理。

#### Aodv\_rreq.c

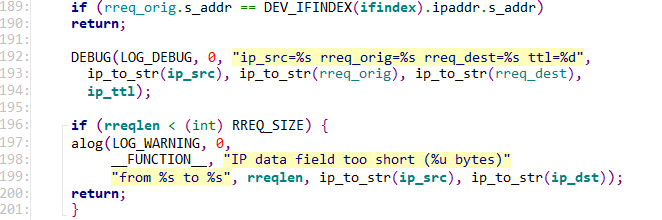
##### 全局变量

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 作用 |
| rreq\_gratuitous | int | 判断是否为免费路由 |
| expanding\_ring\_search | int | 判断是否使用扩展环搜索 |
| internet\_gw\_mode | int | 判断子网位置 |
| wait\_on\_reboot | int | 判断是否在等待重启 |
| hello\_qual\_threshold | int | 判断是HELLO消息否符合门槛 |
| ratelimit | int | 判断是否启用了速率限制 |
| gw\_prefix | int | 判断地址是否加了前缀 |

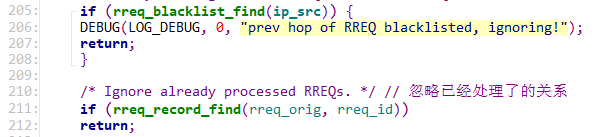
##### 代码分析



* 函数创建RREQ，初始化信息的类型，设置该信息的地址、序列号等信息。

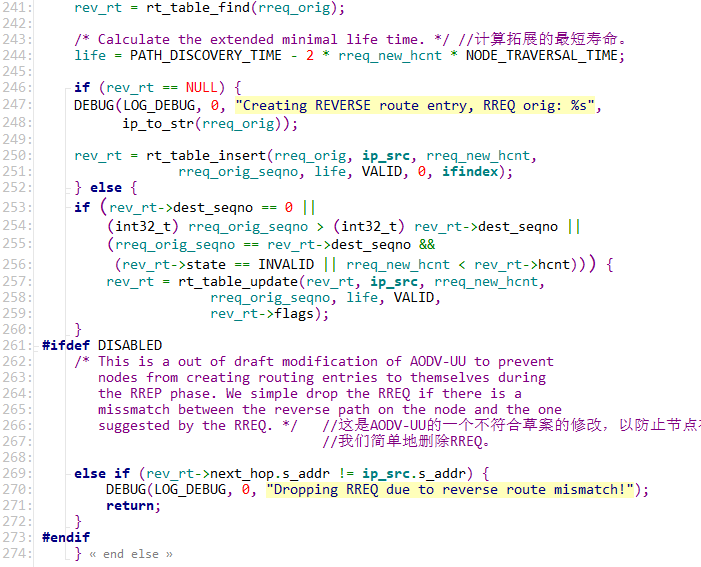


189~194：检查接收到RREQ是否为自身节点发出，如果是，则忽略，如果不是，接收信息。



205~208：检测RREQ的消息是在黑名单中，如果在，则忽略。

210~212：检测该消息是否已经被处理。



241：判断是否存在反向路由表项。

246~248：若不存在反向路由表项，则新建反向路由。若存在反向路由表项，判断是否需要更新。

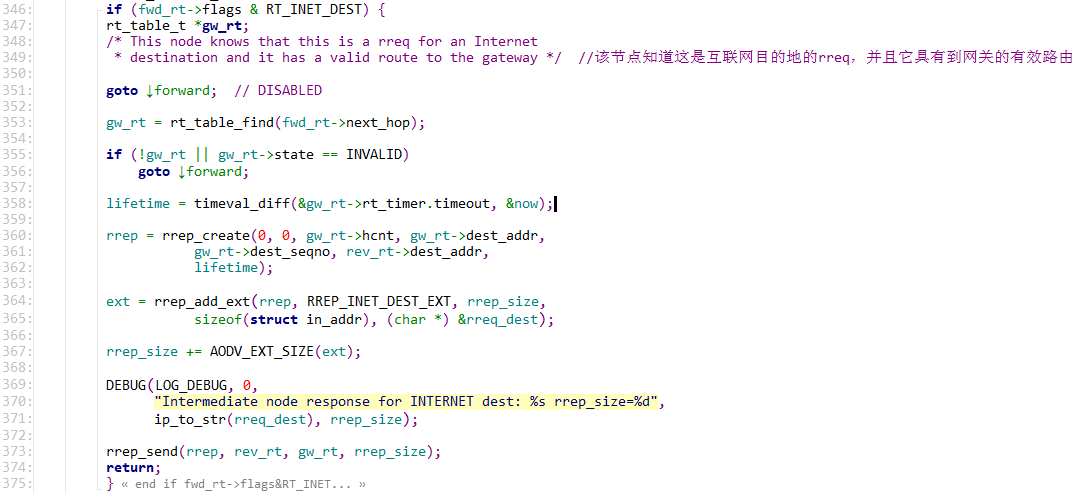
252~260：路由判断，当遇到新的RREQ时，比较在路由表中已经存在的到目的节点的路由，比较目的节点的序列号，在目的节点序列号相同的情况下，采用跳数小的。



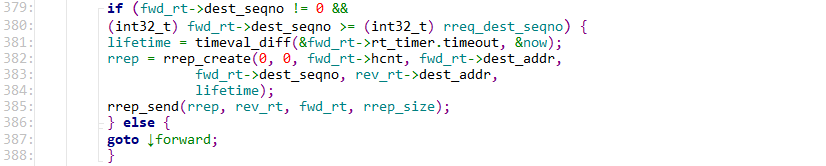
313：当不知道目的节点的位置时，判断自身是不是RREQ的目的节点。

318~329：当节点自身是RREQ的目的节点，将系欸但自身的序列号进行更新，将其更新为当前序列号和RREQ中最大的序列号，并创建新的到该节点的RREQ路径，并开始往回发送RREP消息。

330~338：若不是目的节点，则检查该节点所有的路由条目，查看是否有到达目的节点的路由条目。



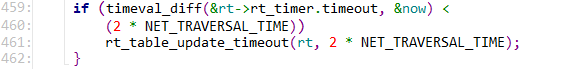
346~375：判断当前节点是否有到目的节点的有效路由，若是则回复RREQ消息，倘若没有则继续广播RREQ消息。



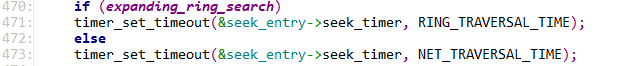
379-388：判断接收到的RREQ消息的序列号是不是足够的新，若不是说明是过时的消息，则不用转发。



437~439：如果当前节点的路由表中已经包含了目的节点的路由表项，则讲ttl设置为NET\_DIAMETER。



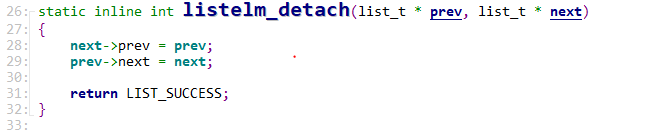
459~460：等待RREP的路由表条目不应该在2\*NET\_TRAVERSAL\_TIME遍历时间之内被删除，若时间小于2\*NET\_TRAVERSAL\_TIME，更新它的计时器，防止被删除。



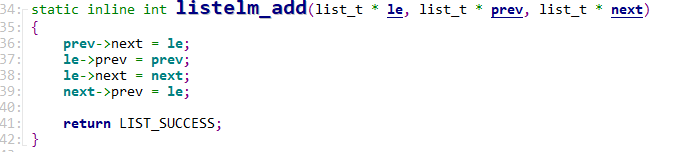
470~473：为RREQ消息设置定时器。

#### List.c分析

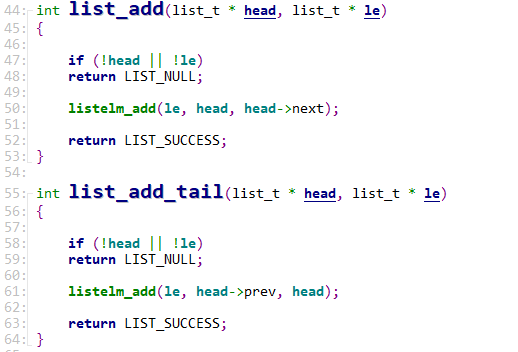
List.c中主要包含了一些链表的操作。



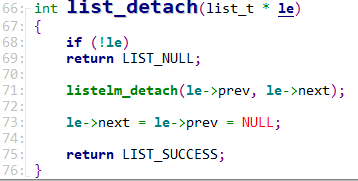
26~34：这个函数进行的是链表中间节点删除的操作。



34~42：这个函数所进行的是在链表中间入新节点的操作。

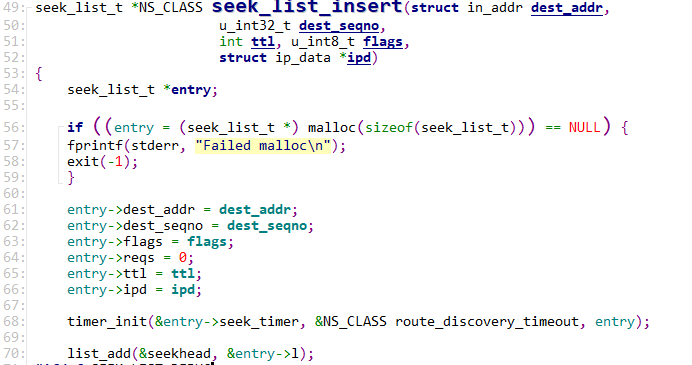


44~64：这两个函数分别实现类的在链表头和链表尾插入新的节点。

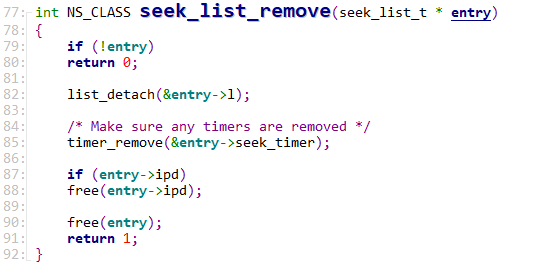


66~76：将链表中的某一指定节点删除。

#### Seek\_list.c分析

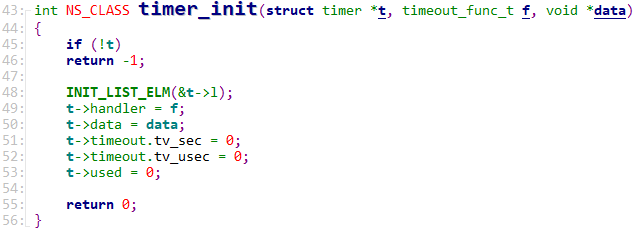


49~70：函数seek\_list\_insert完成的操作是在搜索裂变中加入新的节点。先开辟一个新的节点空间，将地址、序列号，ttl，ip等信息存放进去，并设置计时器，最后将其插入到链表中。

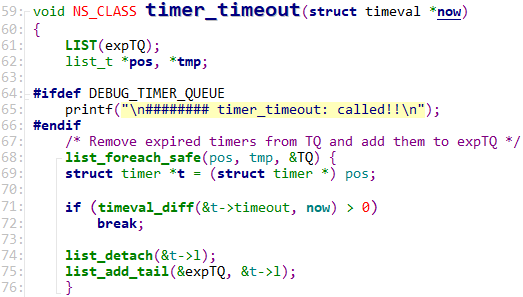


77~92：函数seek\_list\_find完成的操作是将某一结点从搜索链表中删除。先判断该节点是否存在，当节点存在时，删除该节点，并将该节点所占用的空间释放。

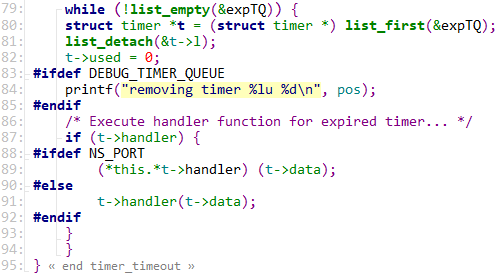
#### 2.2.10 timer\_queue.c



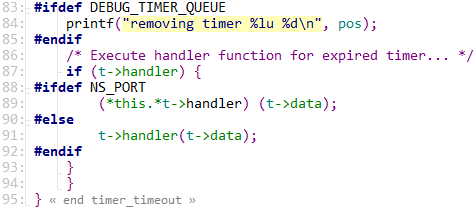
43~56：函数实现定时器的初始化，主要设置定时器的处理函数（handler）、数据（data）、定时器超时秒（tv\_sec）、定时器超时微秒（tv\_usec）、使用状况（used）。



68~75：遍历定时器队列，找到超时的定时器，将其从定时器队列中移除并添加至过期定时器队列的尾部。



79~85：从头部安全地将过期定时器从过期定时器队列中移除，并发出提示信息“removing timer \*\* \*”。



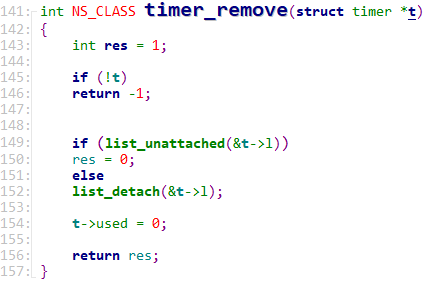
87~95：执行过期计时器的处理函数。



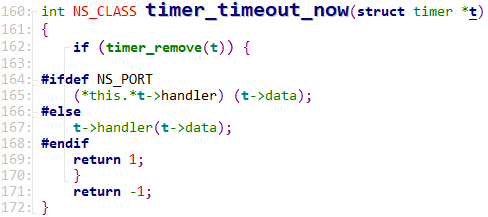
102~109：若定时器为空，则发出错误提示“NULL timer!!!”并终止；若定时器的处理函数为空，则发出错误提示“NULL handler!!!”并终止。

112~113：在添加新的超时之前，删除未过期的计时器。

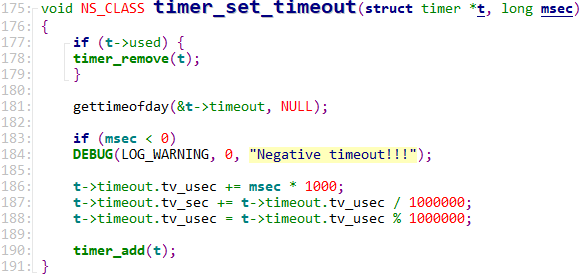
122~133：若定时器队列为空，则直接添加至定时器表中；否则，遍历整个定时器队列，找到第一个超时的定时器的位置，将新的定时器添加至其前面。



141~157：函数实现一个定时器的移除过程，首先判断是否存在该定时器，若否则返回-1，若有，则判断该定时器是否在定时器队列中，若否，则无法移除，若是，移除，将其使用状况置为0，并返回1。

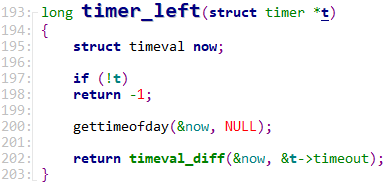


160~172：函数实现被移除定时器处理函数的调用，若成功移除定时器，则调用其处理函数并返回1，否则直接返回-1；

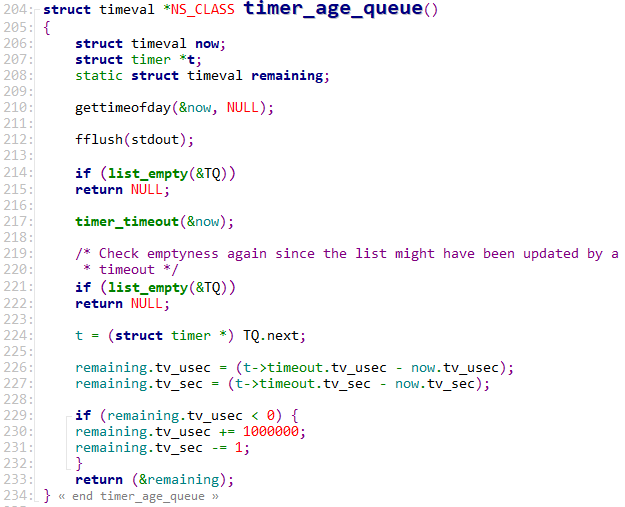


177~179：若该定时器正在计时，则移除它。

183~190：若设置超时为负数，则发出提示信息“Negative timeout!!!”，否则，根据秒-微秒转换设置超时，并将其加入定时器队列。



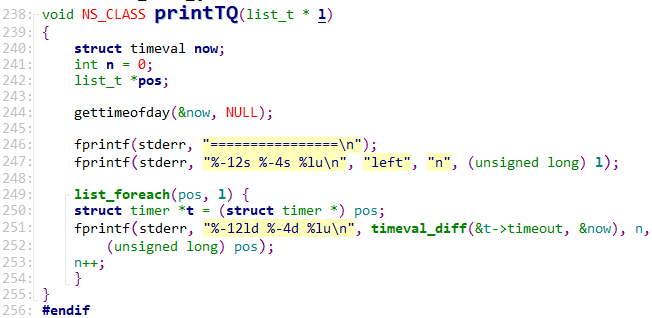
193~203：函数实现查询定时器超时，首先若定时器不存在则直接返回-1，若是，得到当前的时间，计算返回定时器中存放的时间与当前时间的时间差。



214~215：若定时器队列为空，直接返回。

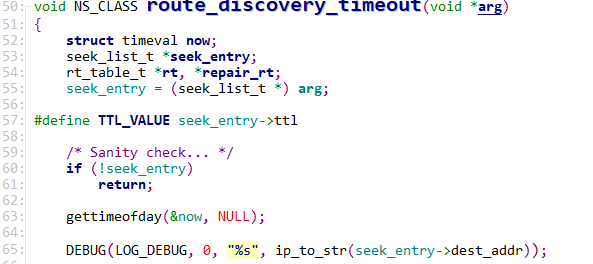
221~222：若移除过期定时器后定时器队列为空，直接返回。

224~233：计算并返回定时器队列中的第一个定时器超时。



238~255：函数实现打印定时器队列，首先调用gettimeofday(&now,NULL)得到当前时间，再遍历定时器队列，计算每个定时器超时剩余时间并打印。

### 2.2.2 Aodv\_timeout.c分析



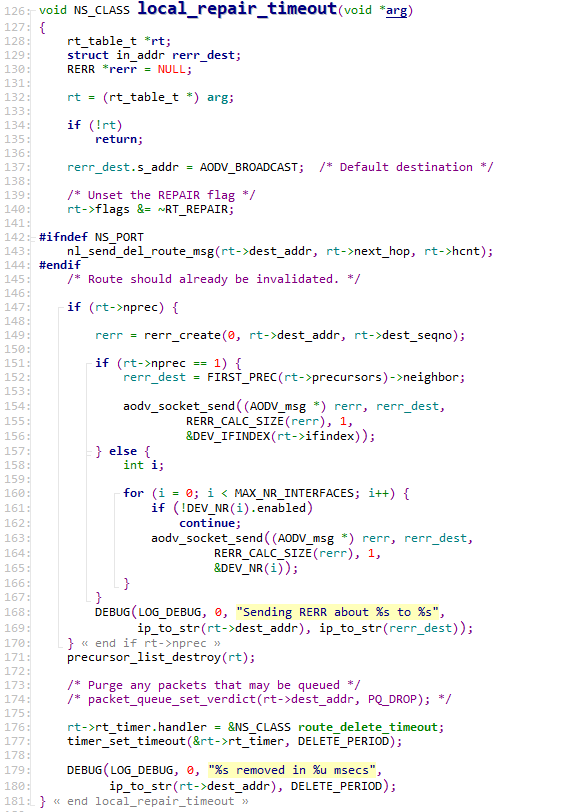
50~65：在函数route\_discovery\_timeout中，获取当前时间，并将其记录到日志中。若该节点为空，则忽略。



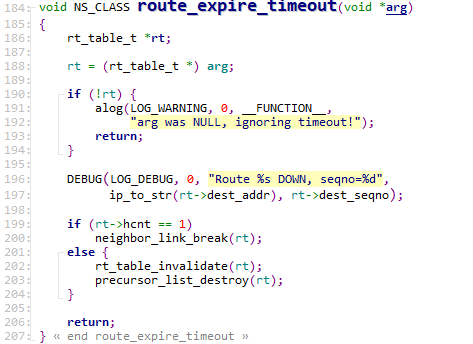
67~103：如果还不满足超时要求，就设定一个新计时器来记录路由发现所用的时间，并把时限设置为 2\*NET\_TRAVERSAL\_TIME。



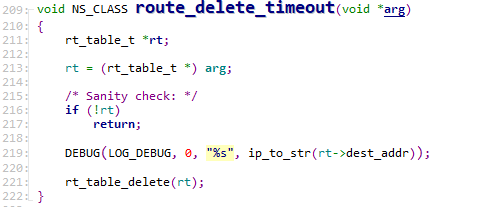
103~124：如果超时就返回没有发现路由信息。 如果此路由已在修复中，那么我们就暂停此路由，并返回超时信号。



126~181：函数local\_repair\_timeout的主要作用是修复超时。广播搜索需要修复的节点，并在找到后删除需要修复的标志位，并将该节点信息作废。创建并广播rrer消息。如果修复超时，就清除队列里所有可能引用处于修复状态的 数据包，并记录到日志里。



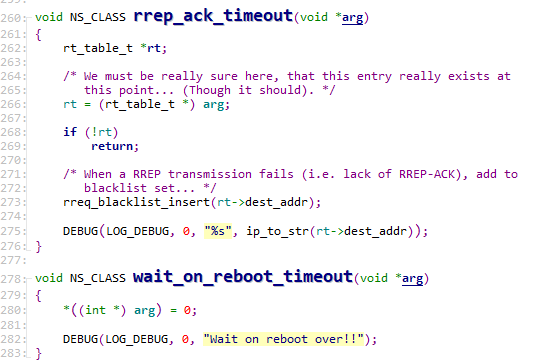
184~207：函数route\_expire\_timeout的作用为检查该节点的信息是否超时，若是已经超时，记录到日志中，切断与邻居节点的关联，并将这个节点的信息删除。



209~222：从路由表里删除该路由信息，并记录日志。

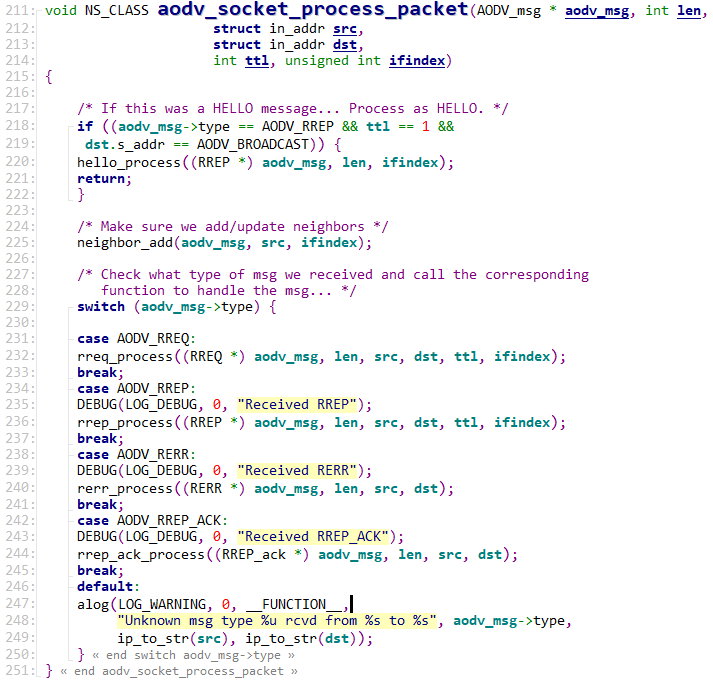


226~258：节点通常会发送HELLO消息给邻居节点确保链路连通，而当我们长时间没收到hello消息时，便调用函数hello\_timeout。获取当前系统的时间。若该路由可以被修复，则设置标志位，并将该节点断开。

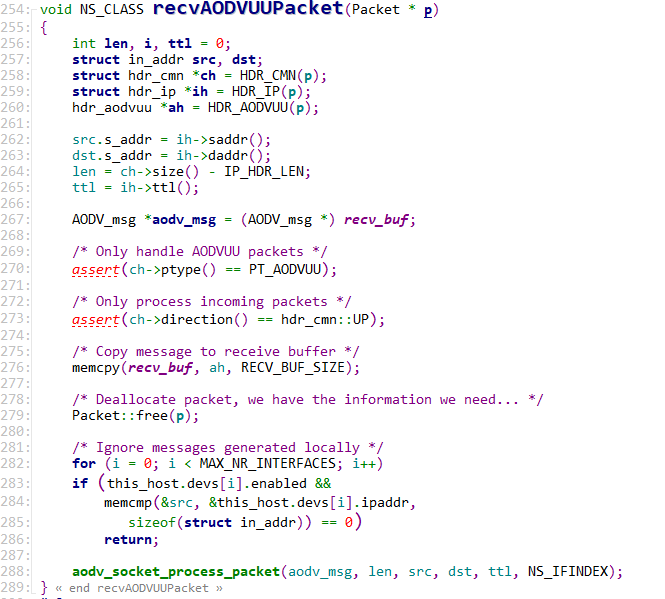


260~283：回复确认超时函数，如果回复传送失败，就把目的地址拉入黑名单。

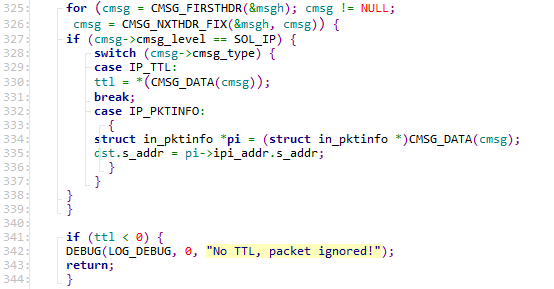
### 2.2.4 Aodv\_socket.c分析



211~251：为函数aodv\_socket\_process\_packet,该函数对所接收到的信息做出判断，判断是属于哪一类的信息，并根据不同的信息做出不一样的处理。

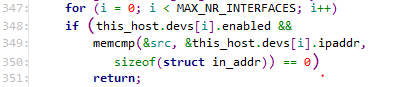


254~289：对接收到的数据包做处理。在确定这是aodv的数据包并且是从外部接受到的数据包时，复制该数据包里的消息到缓冲区，在获取完所需要的数据后，将数据包分解、释放。如果是自身产生的数据包，就将其忽略掉。最后在其做完处理判断后，调用aodv\_socket\_process\_packet对数据包的类型做出相应判断并做相应处理。

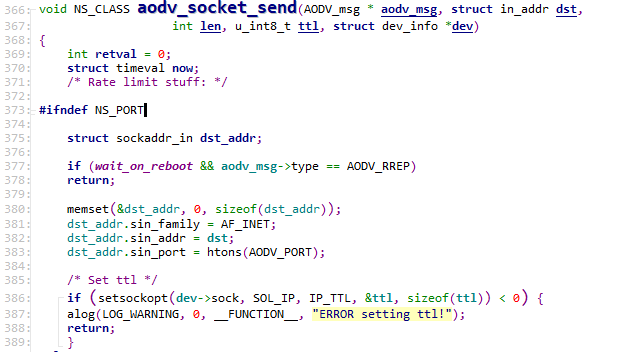


325~339：这段代码在函数aodv\_socket\_read中，这段代码的的作用是从读到的消息中获取ttl（跳数）以及目的地址的信息。

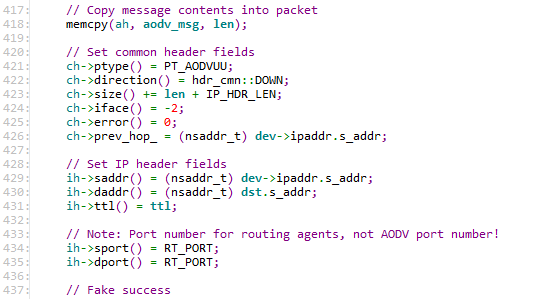
341~344：根据读到的ttl做判断，如果ttl小于0，则忽略该条消息。



347~351：对消息做出判断，检查这条消息是不是自身发出的，如果是自身发出的消息，则将其忽略。

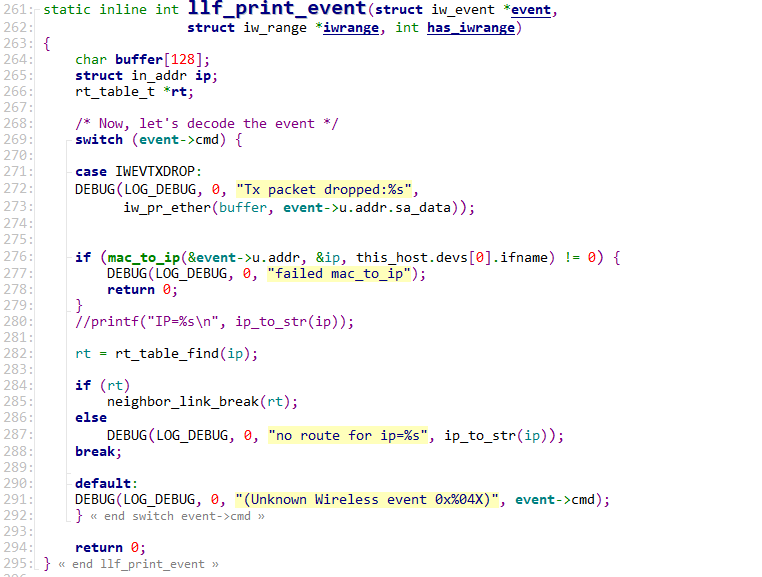


366~389：在函数aodv\_socket\_send中，先判断要发送的信息的类型，先判断信息是否在等待被发送以及信息的类型，若信息不是RREP，则设置该信息的基础信息，并设置ttl。



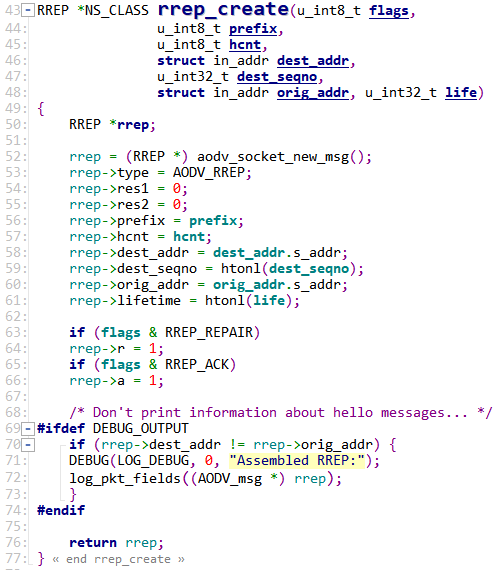
417~437：设置其他首部信息，并设置接收方的端口号。

### 2.2.6 Locality.c分析

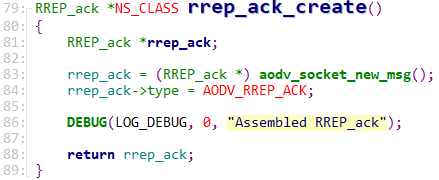


261~295：函数llf\_print\_event实现把打印事件转码并记录到日志里。

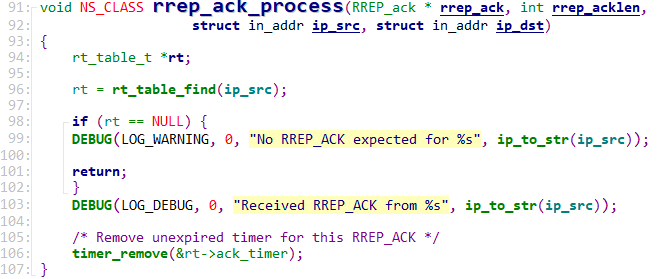
### 2.2.7 aodv\_rrep.c



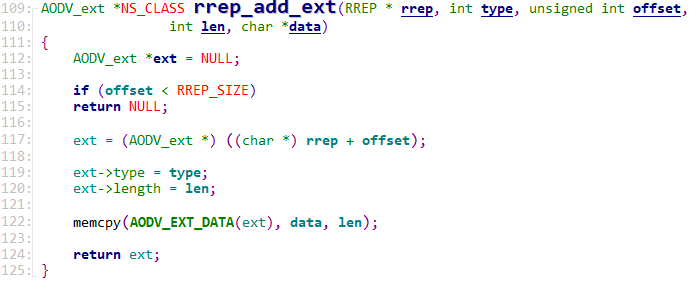
43~77：函数创建RREP消息，初始化信息的类型，设置其前缀长度（prefix）、跳数（hcnt）、目标节点地址（dest\_addr）、目标节点序列号（dest\_seqno）、发起RREQ节点地址（orig\_addr）以及生命周期（lifetime）。



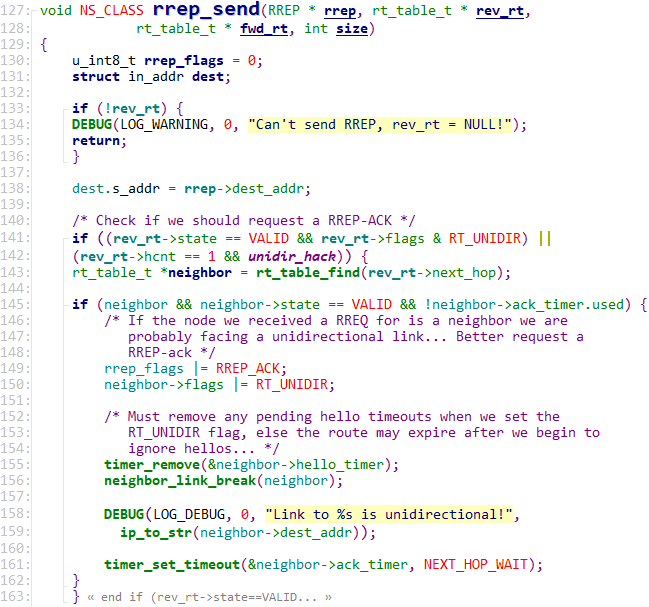
79~89：函数创建RREP\_ACK消息，并发出提示信息“Assembled RREP\_ACK”。



94~106：查找该结点的路由表，判断是否有与接收到的RREP\_ACK消息中一致的源IP地址，若rt == NULL则表示该节点中没有到该地址的路径，因此接收到的 RREP\_ACK 消息非法，直接返回。若包含相同的IP地址，则该节点有到达RREP\_ACK中源节点的路径，移除路由表中关于该RREP\_ACK的未失效的定时器。



109~122：函数创建RREP消息的扩展，首先判断位移是否小于一个基本首部长度，若是，则扩展部分与基本首部有重叠，则该扩展非法，返回NULL；若否，则使指向aodv\_ext 的指针ext指向RREP基本首部后第一个字节，并设置其类型（type）、数据长度（length）和数据（data）。

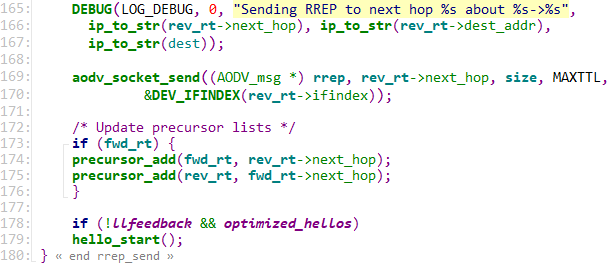


133~136：首先判断该节点是否存在反向路由表，若无则无法发送RREP消息。

141~143：检查是否需要请求一个RREP\_ACK消息。

145~150：若收到的RREQ消息中的目的节点地址是该节点的邻居节点时，则有可能要面临单向连接的情况，此时最好请求一个RREP\_ACK消息。

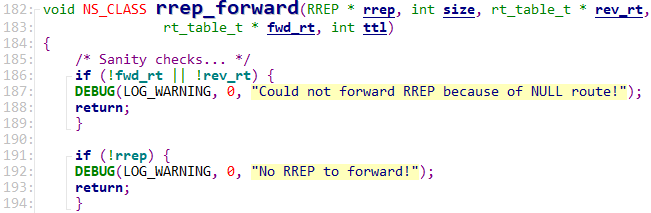
155~161：移除在设置单向连接期间内的所有hello定时器，否则在开始忽略hello时路由可能会失效。发出提示信息“Link to \*\*\* is unidirectional!”并设置邻居节点的ACK定时器为下一跳等待时长。



165~170：发出提示信息“Sending RREP to next hop \*\* about \*\*->\*\*”并建立套接字发出RREP消息。

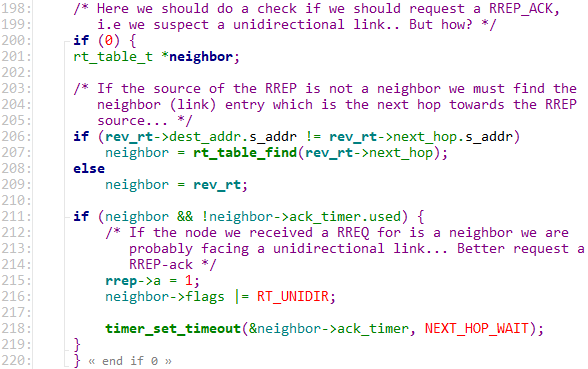
173~176：若该节点存在正向路由，则更新该节点的前驱表。

178~179：若未开启链路层反馈，且只有在转发数据时才发送hello消息时，启动发送hello消息。



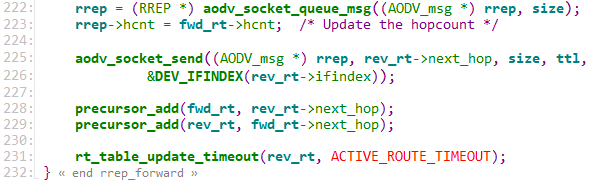
186~189：若该节点正向路由或反向路由为空，则发出提示信息“Could not forward rrep because of NULL route!”并返回。

191~194：若RREP消息为空，则发出提示信息“No rrep to forward!”并返回。



206~209：判断该RREP消息中的源节点是否是该节点的邻居节点，若不是，则查找通向源节点的下一跳邻居节点。

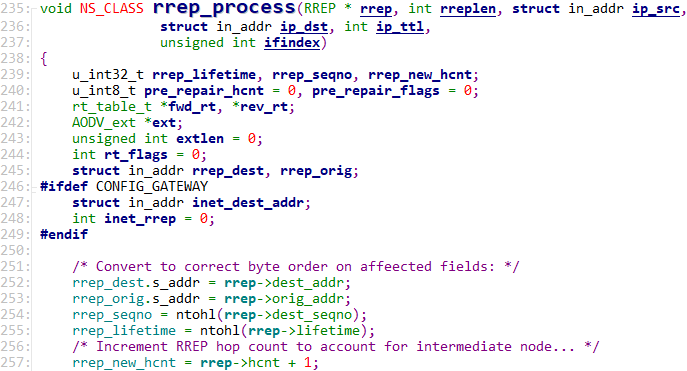
211~218：若收到的RREQ消息中的目的节点地址是该节点的邻居节点时，则有可能要面临单向连接的情况，此时最好请求一个RREP\_ACK消息并设置邻居节点的ACK定时器为下一跳等待时长。



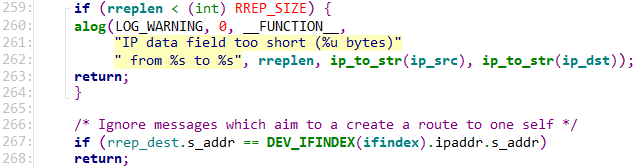
222~226：更新RREP消息中的跳数并通过socket转发。

228~229：更新该节点的前驱表。

231~231：更新该节点的反向路由中的定时器。

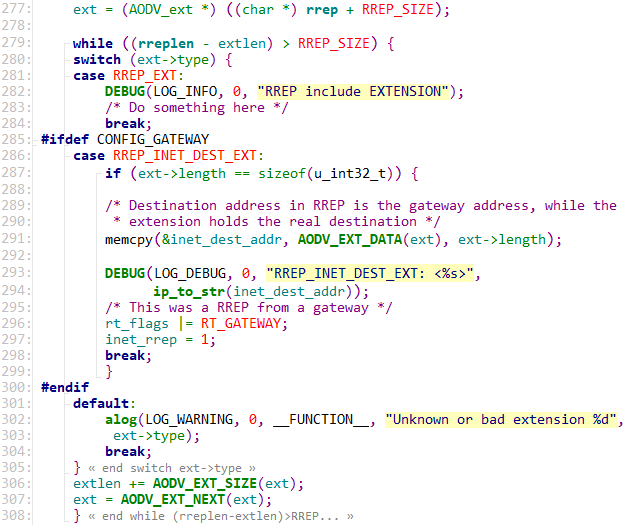


235~257：声明相关局部变量，并将RREP中的信息由网络字节序转换为本机字节序。



259~264：判断收到的RREP消息长度是否小于RREP消息最小长度，若是，则该RREP消息无效，发出提示信息“IP data field too short (\* bytes) from \*\* to \*\*”并返回。

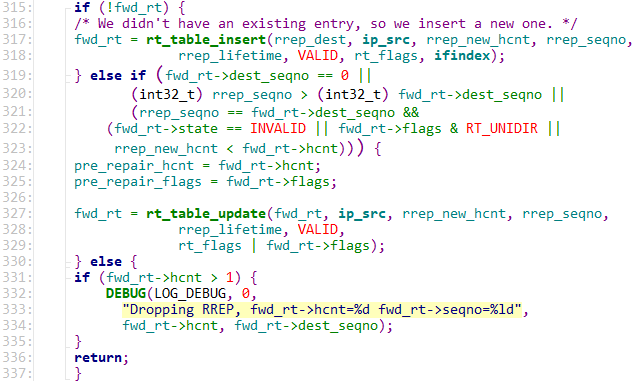
267~268：忽略到自身建立路由的RREP消息。



281~284：若该RREP消息中含有扩展，则发出提示信息“RREP include EXTENSION”。

285~299：若该RREP中的目的节点地址是网关地址，则其扩展中存有真正的目的节点，则将扩展中的内容复制到inet\_dest\_addr变量中并发出提示信息“RREP\_INET\_DEST\_EXT:<\*\*\*>”并设置相关标志位。

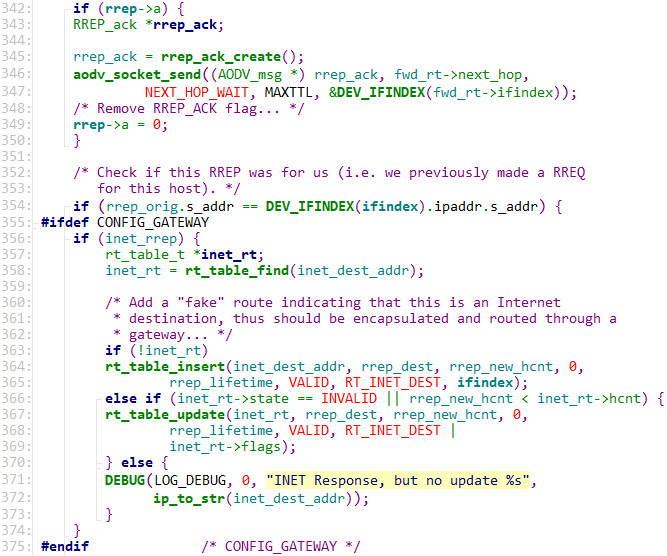
306~307：更新扩展长度及其指针指向位置。



315~318：若该节点正向路由表项为空，则向该节点插入一个正向路由表项。

319~329：若该表项存在，且路由表中序列号在路由表项中被标示无效，或路由表项序列号大于目的节点序列号在该节点的副本，或序列号相等，但被路由标识为无效，或序列号相同且新跳数小于路由表项跳数，则更新转发路由中的条目。

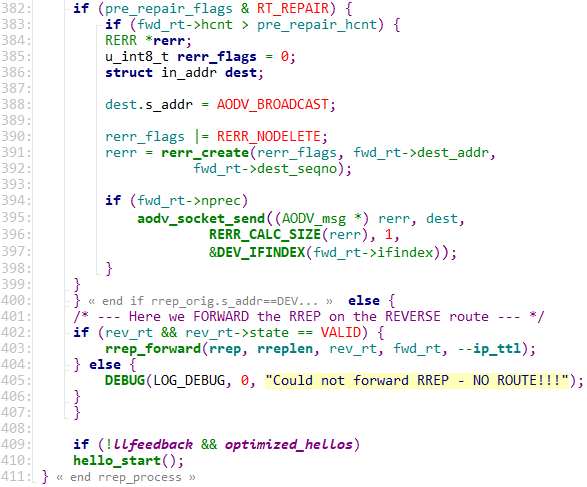
330~336：上述条件不满足，则丢弃该RREP信息。



342~350：若该RREP消息中的A标志位置1，则创建并向下一跳节点发出一个RREP\_ACK消息，并将该RREP消息中的A标志位置0。

354~354：判断该RREP消息是否是该节点的路由回复消息。

355~375：若该节点的RREP的目的节点是网关地址，则查找路由表是否有到达该地址的路径，若无，则插入一个“伪”路由；若有，且RREP中的跳数小于该路由表项中的跳数，则更新该路由表项，否则发出提示消息“INET Response, but no update \*\*”。

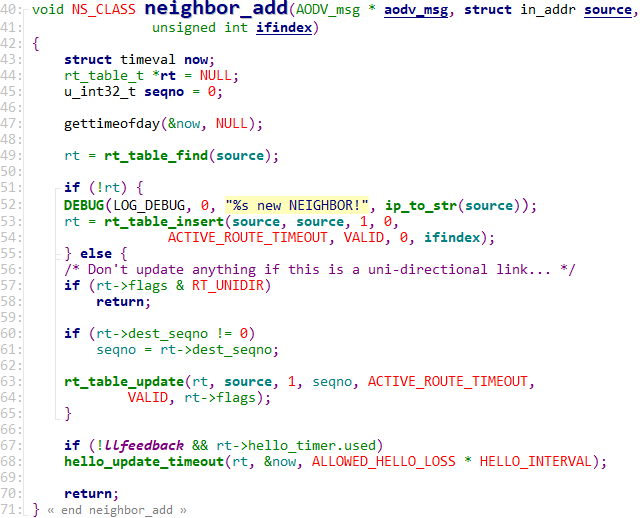


382~399：如果路由之前处于修复状态，则应创建RERR消息，将NO DELETE RERR发送到路由的源节点，以便它可以选择重新启动目的地的路由发现。

402~407：通过反向路由转发一个RREP消息。

409~410：若未开启链路层反馈，且只有在转发数据时才发送hello消息时，启动发送hello消息。

### 2.2.8 aodv\_neighbor.c

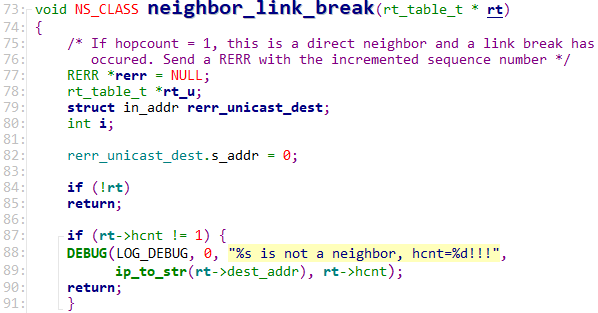


49~54：查找路由表中是否有到hello消息发送方的路径，若没有，则发出提示信息“\*\* new NEIGHBOR!”并在路由表中新建该路由。

57~58：若路由表中有到hello消息发送方的路径，且该路径为单向连接，则不更新该路径的信息。

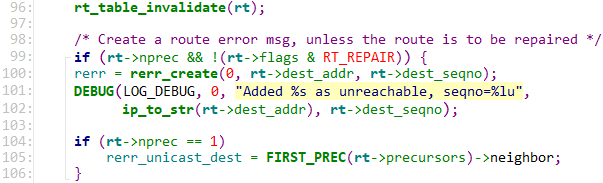
60~61：若该路径上的目的序列号不为0，则更新路由表中该路径的目的序列号，并重置活跃路由定时器时间。

67~68：若未启动链路层反馈，且该路径上已发送过hello消息，则其hello消息定时器增加时间ALLOWED\_HELLO\_LOSS \* HELLO\_INTERVAL。



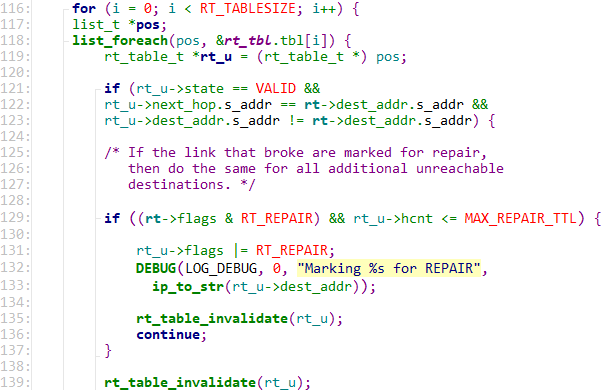
84~85：若路由为空，则并非链路中断，直接返回。

87~90：若该路径上的跳数不为1，则目的节点并非该节点的直接邻居节点，发出提示消息“\*\* is not a neighbor, hcnt=\*!!!”并返回。



96~96：使该路径的入口无效化。

99~105：若该路径上有前驱表，且并未处于路由修复状态，则创建一个RERR消息，并发出提示信息“Added \*\* as unreachable, seqno=\*”；若该路径上仅有一张前驱表，则RERR消息单播的目的节点地址即是该前驱表内的前驱。

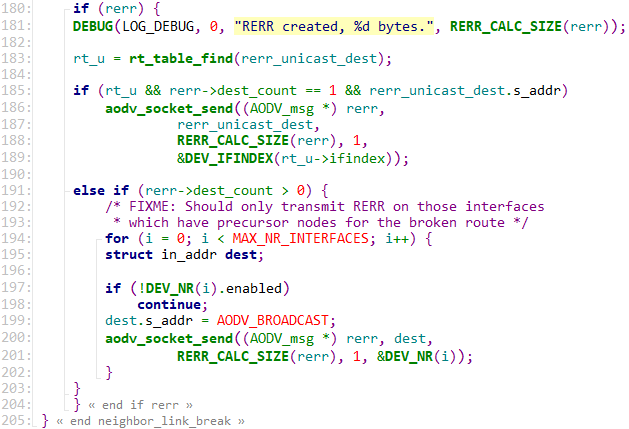


116~139：查找整个路由表，定位到每个路由表项，若该路由的下一跳节点是不可达的，且若中断路由被标志为修复状态,则对查找到的其他目的节点不可达的路由与中断路由一样使路径的入口无效化。



141~153：若下一跳为不可达的路由存在前驱表，且尚未创建RERR消息，则创建一个RERR消息，若该路由的前驱表有且只有1张，则RERR消息单播的目的节点地址为该前驱表中的前驱。发出提示信息“Added \*\* as unreachable, seqno=\*”。

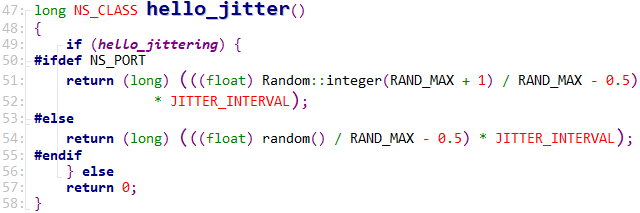
157~169：若已创建RERR消息，则添加该路径上的目的节点地址和目的节点序列号到rrer消息中。若存在RERR单播目的节点地址，则遍历该路径上的前驱表，若存在前驱表项，其邻居节点地址不等于RERR单播目的节点地址，则删除单播目的节点地址。



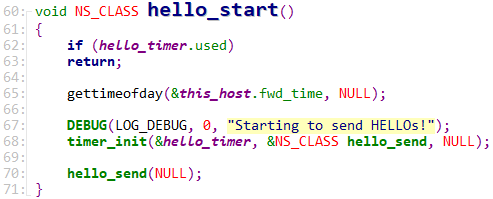
180~189：遍历整个路由表结束，若存在RERR消息，则发出提示信息“RERR created, \* bytes.”并定位目的节点地址为rrer单播目的节点地址的路径。若路径存在，RERR消息目的节点地址数为1，则发送rrer消息。

191~203：若上述条件有不满足的，且rrer消息的目的节点地址数大于0，则应该仅在具有中断路由的前驱节点的接口上传输rrer消息。

### 2.2.9 aodv\_hello.c

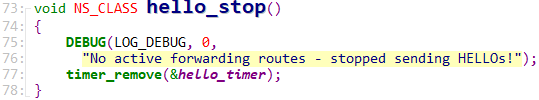


47~57：若hello\_jittering置1，则返回hello消息的抖动时间。

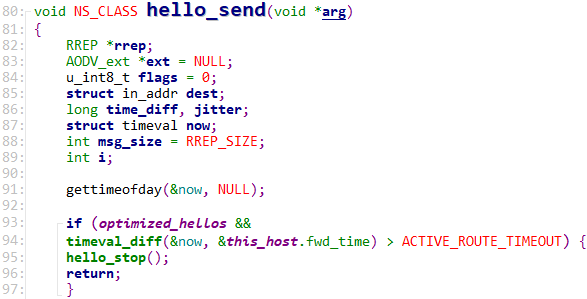


62~63：若hello消息已启动，则直接返回。

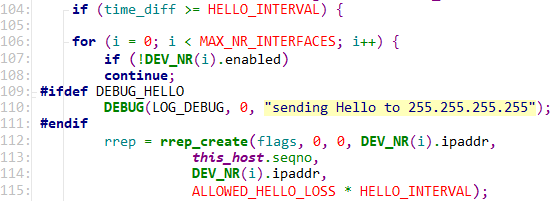
65~70：发出提示信息“Starting to send HELLOs!”，启动hello定时器并发送hello消息。



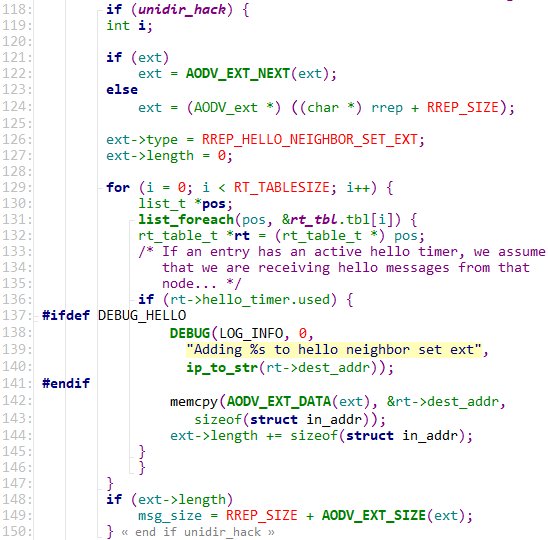
75~77：发出提示信息“No active forwarding routes - stopped sending HELLOs!”并移除hello定时器。



93~96：若只有在转发数据时才发送hello消息，且发出hello消息时间已超过活跃路由定时器时长，则中断发送hello消息，并返回。

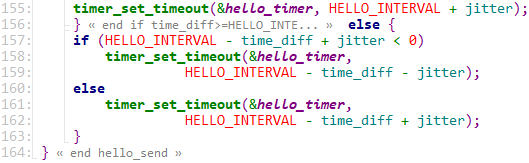


104~115：经过HELLO\_INTERVAL时间，检查在期间是否发出过一个广播，若无则创建一个TTL=1的RREP消息。



121~124：若已存在扩展，则在该扩展后第一个字节开始添加其他扩展；若无，则在基本首部后第一个字节开始添加扩展。

129~150：把hello消息发送方的邻居节点集合装配到hello消息RREP扩展并更改扩展长度与消息长度。



156~162：若hello消息发送时间未超过一个HELLO\_INTERVAL，则调整hello定时器。

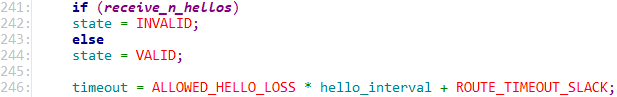


170~189：声明相关局部变量，将收到的hello消息中的相关数据从网络字节序转换为本机字节序，并查找路由表中是否有到达hello消息中的目的节点的路径，若有则根据路径属性修改相关局部变量。

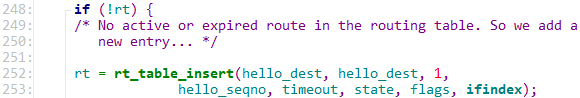


192~208：若hello消息长度大于RREP最小长度，判断扩展类型，若为RREP\_HELLO\_INTERVAL\_EXT，且扩展长度为4，则将扩展的内容复制到hello\_interval。

209~221：若扩展类型为RREP\_HELLO\_NEIGHBOR\_SET\_EXT，则发出提示信息“RREP\_HELLO\_NEIGHBOR\_SET\_EXT”并遍历扩展，若找到与本机相同的地址的内容，则修改flags。



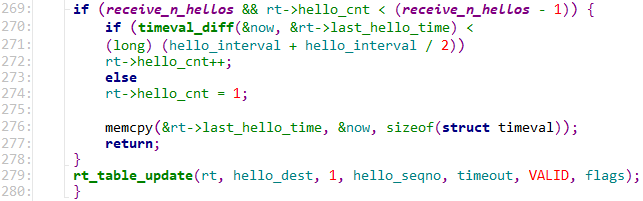
241~244：若邻居节点连续收到3个hello消息，则将其设为有效。



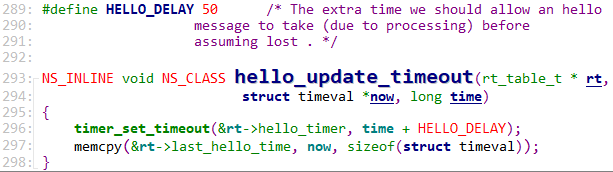
248~253：若路由表中没有活动路由或过期路由，则添加一个hello消息传播路径。



265~257：若该路径标志为RT\_UNIDIR，且路径有效，跳数大于1，则更新hello消息的定时器，至少为ALLOWED\_HELLO\_LOSS \* HELLO\_INTERVAL。

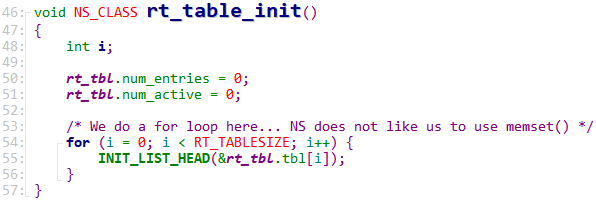


269~278：对于可通往临近结点的活跃路由，通过hello消息更新到最新的目的序列号。

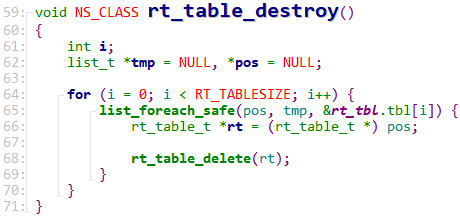


289~298：由于网络处理延迟，宏定义HELLO\_DELAY设置hello消息允许的延时为50。

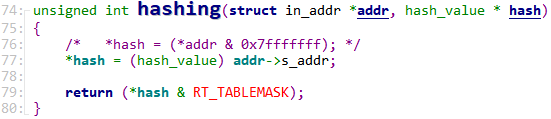
### 2.2.11 routing\_table.c



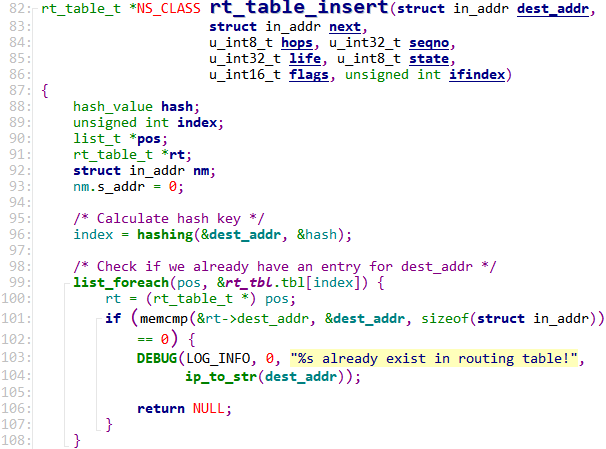
46~57：函数实现路由表的启动，初始化路由表中的路由表项与活跃路由数为0，再遍历路由表创建每个路由表项的头。



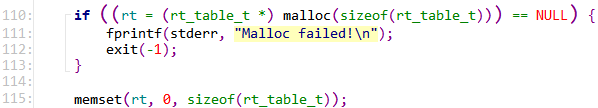
59~71：函数实现路由表的清除工作，安全遍历整个路由表，调用rt\_table\_delete()移除所有路由表。



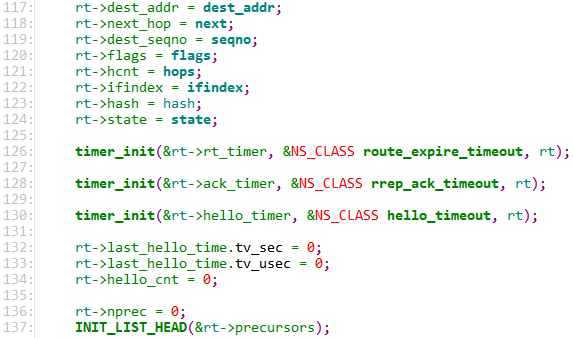
74~80：函数实现计算输入的地址（键）的哈希值和表索引。



96~107：得到输入目的节点地址的索引，并遍历对应路由表，若发现已有对应的路由表项，则发出提示信息“\*\* already in routing table!”并返回NULL。

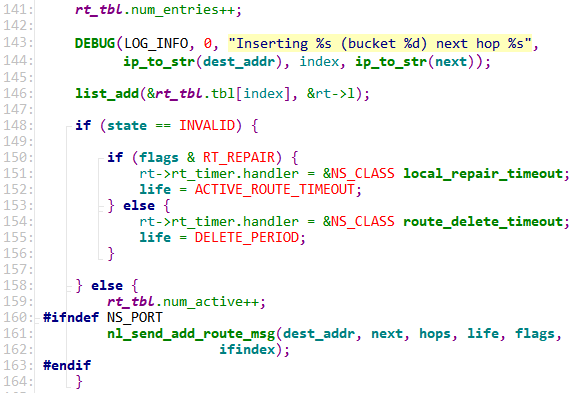


110~115：开辟一个存放路由表项的空间，若开辟失败，发出错误提示“Malloc failed!”并退出；开辟成功，则初始化空间为0。



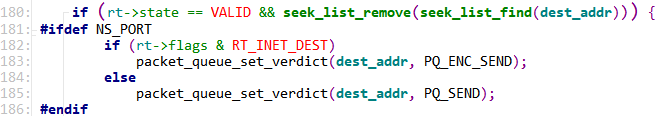
117~124：初始化路由表项，为目的节点地址（dest\_addr）、下一跳地址（next\_hop）、目的序列号（dest\_seqno）、标志位（flags）、跳数（hops）、前缀长度（ifindex）、哈希值（hash）、路由状态（state）赋值。

126~134：启动相关计时器并初始化超时。

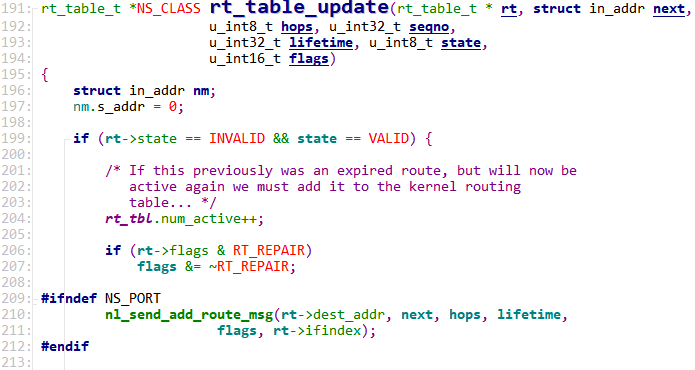


143~146：发出提示信息“Inserting \*\* （bucket \*）next hop \*\*”并向路由表中插入该路由表项。

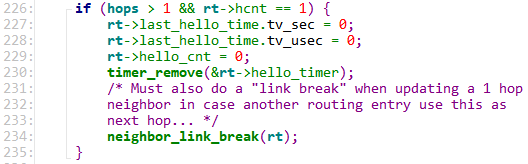
148~159：若路由表项状态为无效，且处于路由修复时，调用处理函数进行设置修复超时并更新生命周期，若未处于路由修复状态，则调用处理函数设置删除超时并更新生命周期；若路由表项状态为有效，则为活跃路由，路由表活跃路有数加1。



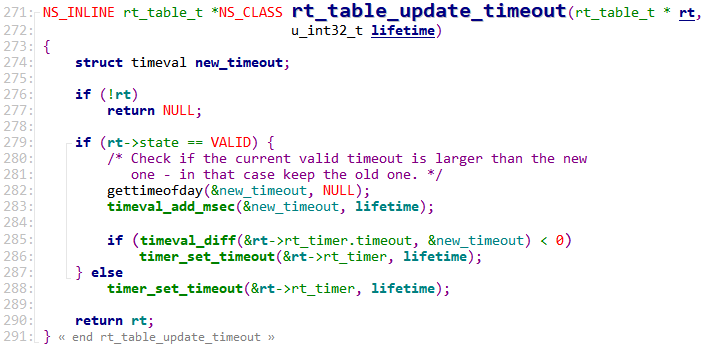
180~186：若目的节点有缓冲包，将这些缓冲包发送到新的路由。



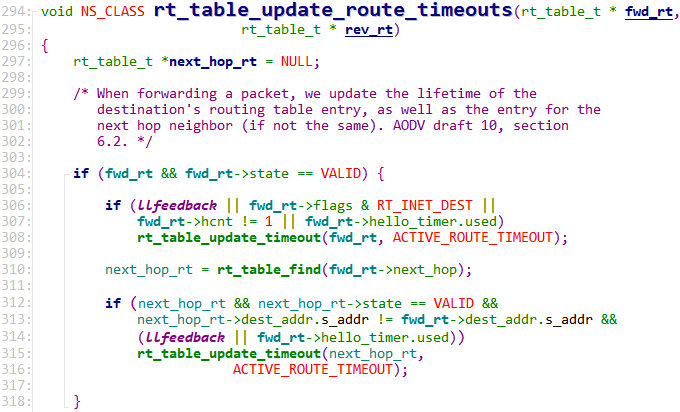
199~207：若该路由原来已经无效，且当前重新活跃起来，则需要将其加入核心路由表。若该路由原来处于路由修复状态，则修改flags位。



226~235：若原路由跳数为1，且新路由跳数大于1，则归零该路由的hello超时并移除hello定时器，且为防止其他路由表项使用其作为下一跳，切断该表项与前后表项的连接。

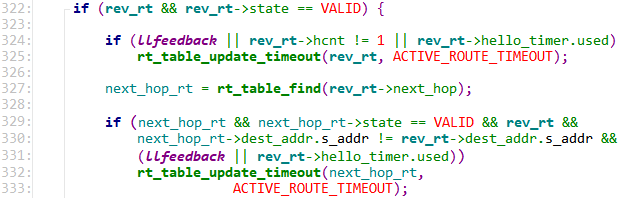


279~288：若该路由为活跃路由，则检查当前有效超时是否比新的超时更短，若是，则更新路由的生命周期，否则保持原生命周期；若当前路由无效，则直接更改生命周期。

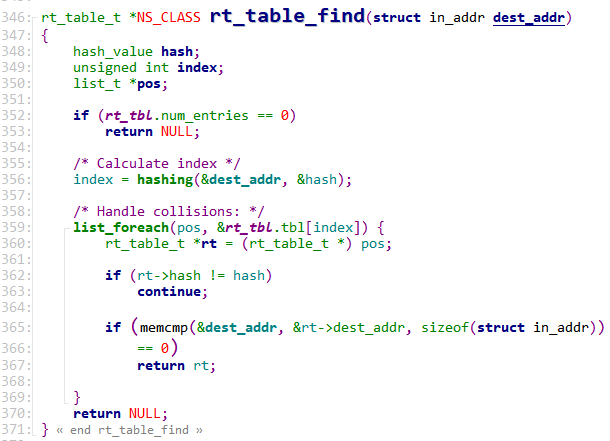


304~308：转发数据包时，若开启了链路层反馈，需要同时更新目的节点路由表项。

310~316：若下一跳邻居节点不是目的节点，则也更改其路由表项。

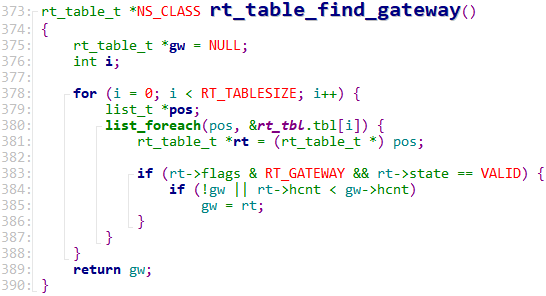


324~333：通过反向路由执行与之前相同的操作，因为路由中的发送方与目的地之间应该是对称的。

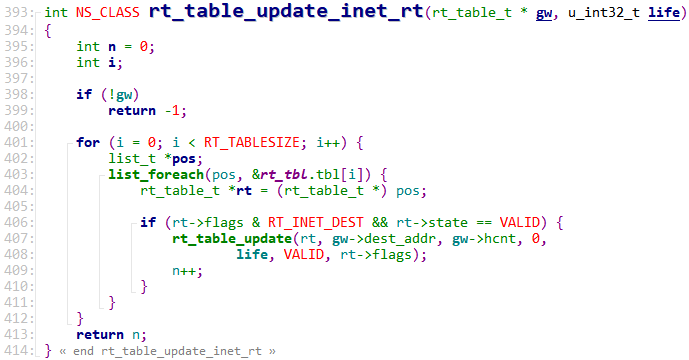


352~353：若路由表中没有路由表项，则返回NULL。

356~370：计算目的节点地址的哈希值，并索引到相关路由表，遍历该路由表直到找到相同的哈希值对应的路由表项，若该表项中的目的节点地址与输入目的节点地址相同，则返回该表项；若遍历到表尾仍未找到则返回NULL。

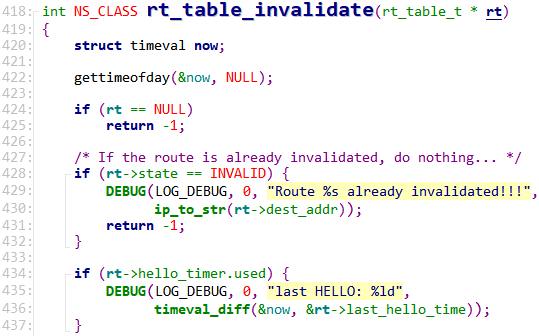


378~388：遍历整个路由表，找到标志位为RT\_GATEWAY，且跳数最大的活跃路由，并返回该路由。



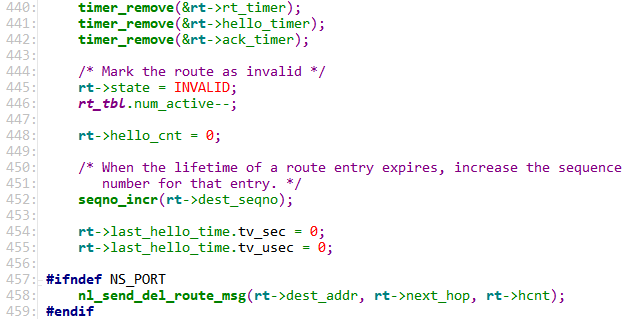
398~399：若网关路由为空，则返回-1。

401~413：遍历所有路由表中的路由表项，若存在标志位为RT\_INET\_DEST的活跃路由，则更新该路由的信息。循环结束，返回更新的路由数。



428~432：若该路由已经失效，则发出提示信息“Route \*\* already invalidated!!!”并返回-1。

434~437：若该路由已发送过hello消息，则发出提示信息“last HELLO：\*”。



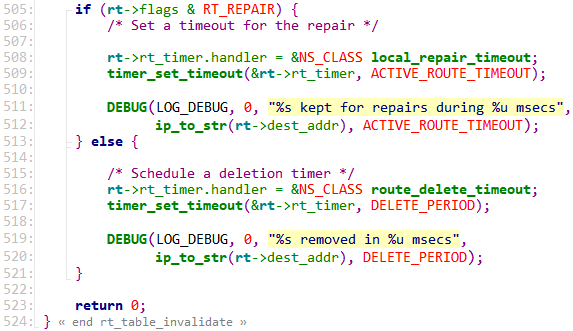
440~442：移除该路由上的定时器。

445~448：该路由状态改为无效。

452~452：若一个路由表项生命周期到了，则增加其序列号。

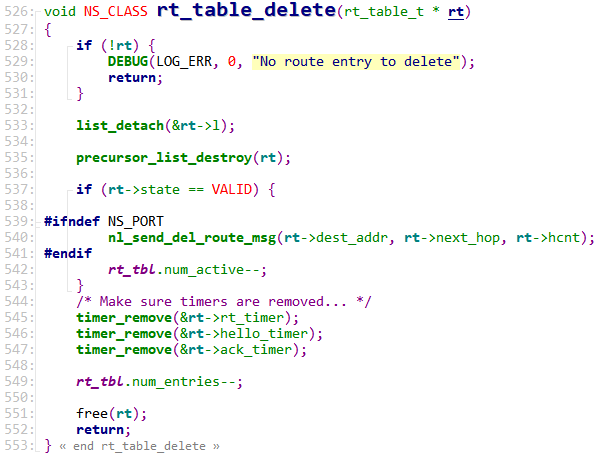


466~497：若该路由为网关路由，则查找路由表中是否有使用该网关的因特网路由，若有且未找到备份网关，则使该路由也无效化；若有备份网关，则更新该路由信息。



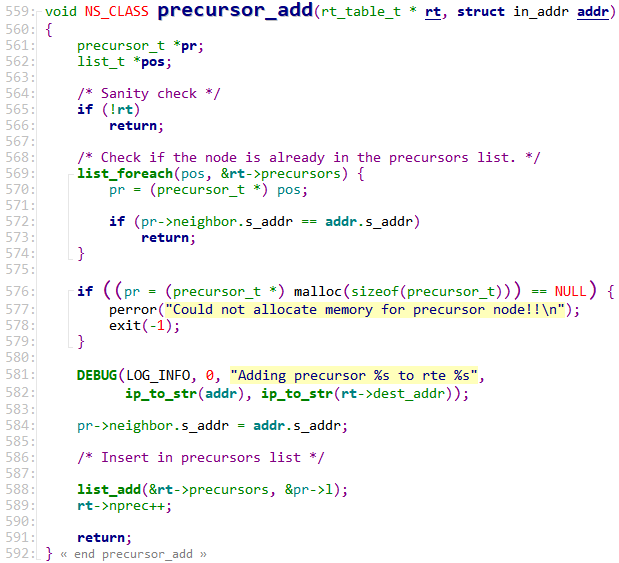
505~512：若该路由标记为路由修复状态，则为路由修复设置一个超时并发出提示信息“\*\* kept for repairs during \* msecs”。

515~520：若未标记为路由修复状态，则设置一个删除超时，并发出提示信息“\*\* removed in \* msecs”。



528~535：若路由表项为空，则发出提示信息“No route entry to delete”；否则，移除该路由中的列表并删除其前驱表。

537~552：若该路由为活跃路由，则移除其定时器并释放该路由在内存中的空间。



565~566：若路由为空，则直接返回。

569~573：查看该节点是否已在前驱表中，若是则直接返回。

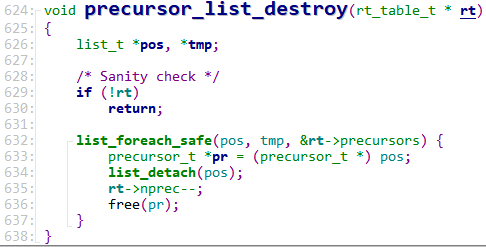
576~578：若为该前驱表项开辟空间失败，则提示错误“Could not allocate memory for precursor node!!”并退出。

581~590：发出提示信息“Adding precursor \*\* to rte \*\*”并将其添加至前驱表。



603~604：若路由为空，则直接返回。

606~616：遍历前驱表，找到对应地址的节点，发出提示信息“Removing precursor \*\* from rte \*\*”并将其从列表中移除，释放该路由空间。



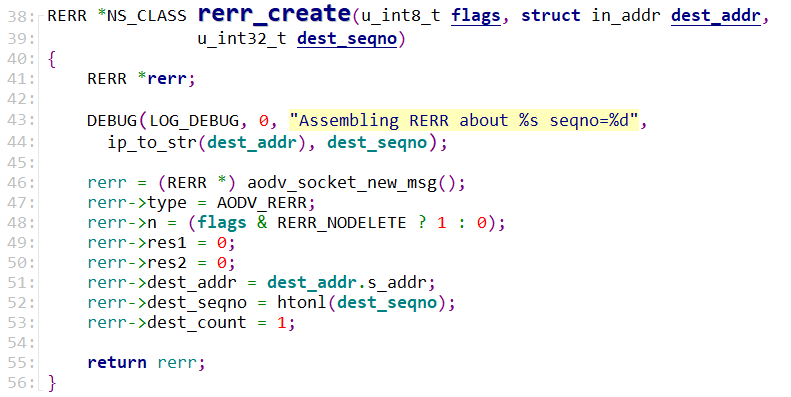
629~630：若该前驱表为空，则直接返回。

632~636：遍历整个前驱表，移除每一个节点并释放其内存空间。

### 2.2.13 aodv\_rerr.c

|  |  |  |
| --- | --- | --- |
| 函数名 | 返回值 | 功能 |
| NS\_CLASS rerr\_create(u\_int8\_t flags, struct in\_addr dest\_addr,u\_int32\_t dest\_seqno) | RERR \* | 创建一条rerr消息 |
| NS\_CLASS rerr\_add\_udest(RERR \* rerr, struct in\_addr udest,u\_int32\_t udest\_seqno) | void | 定义一个RERR\_udest\*类型的变量ud来储存不可达节点的信息，rerr错误信息被更新 |
| NS\_CLASS rerr\_process(RERR \* rerr, int rerrlen, struct in\_addr ip\_src,struct in\_addr ip\_dst) | void | 从rerr消息中得到不可达目的节点，将通往不可达节点的路由rt置为无效路由并最后删除，同时向自己的邻居转播RERR消息 |

RERR \*NS\_CLASS rerr\_create(u\_int8\_t flags, struct in\_addr dest\_addr, u\_int32\_t dest\_seqno):



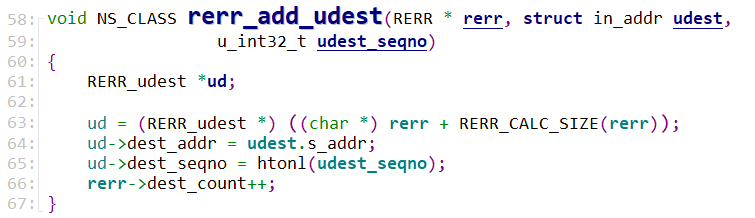
41 创建rerr数据包

43~44 设置输出信息

46 为新产生的数据包分配空间

47~53 给各项参数赋值

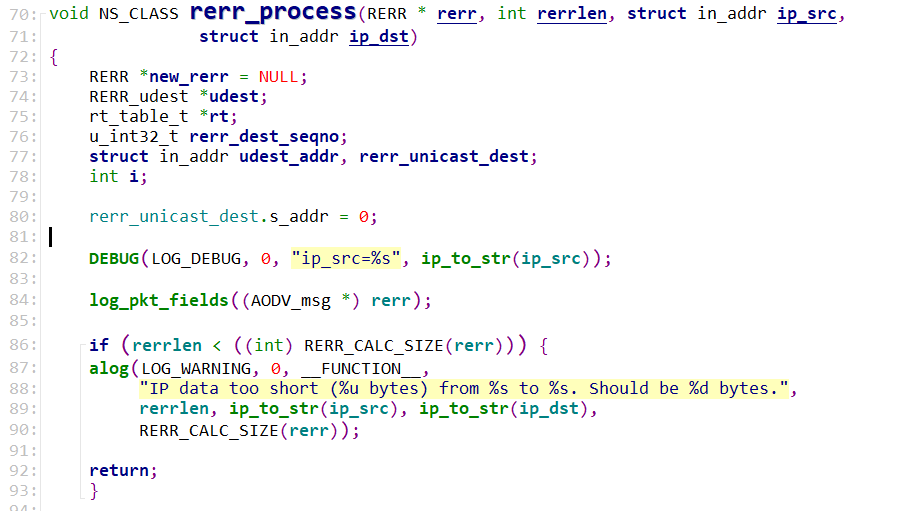
void NS\_CLASS rerr\_add\_udest(RERR \* rerr, struct in\_addr udest,u\_int32\_t udest\_seqno):



61~63 定义一个RERR**\_**udest\*类型的变量ud来储存不可达节点的信息

64~66 ud记录不可达节点的IP地址及序列号等信息，rerr错误信息被更新

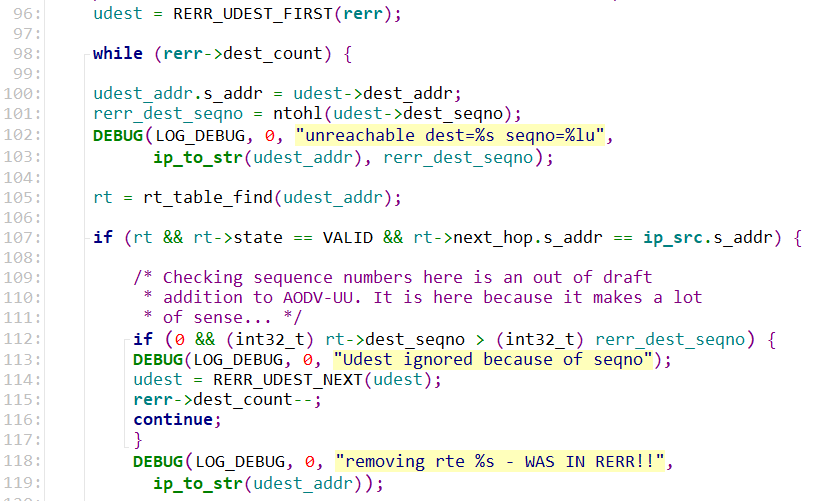
void NS\_CLASS rerr\_process(RERR \* rerr, int rerrlen, struct in\_addr ip\_src,struct in\_addr ip\_dst):



73~80 一些变量的声明、初始化

82 打印调试信息

84 写入日志信息



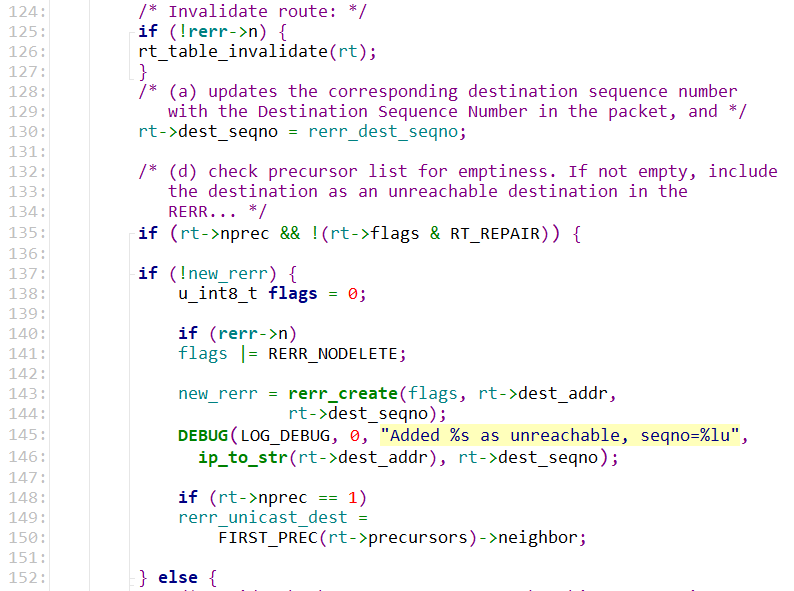
86~92 确定接收到的rerr消息的大小，如果其长度小于下限值则输出错误信息

96~103 从rerr消息中得到不可达目的节点，并对函数中定义的变量赋值，输出调试信息为不可达目的节点的IP地址和序列号

105 寻找到达不可达节点的路由表项

107~117 如果该路由表项仍有效并且下一跳目的节点与发送rerr消息的节点相同则利用目的节点序列号的比较判断消息是否过期，如果是过期消息将忽略

118~119 打印调试信息



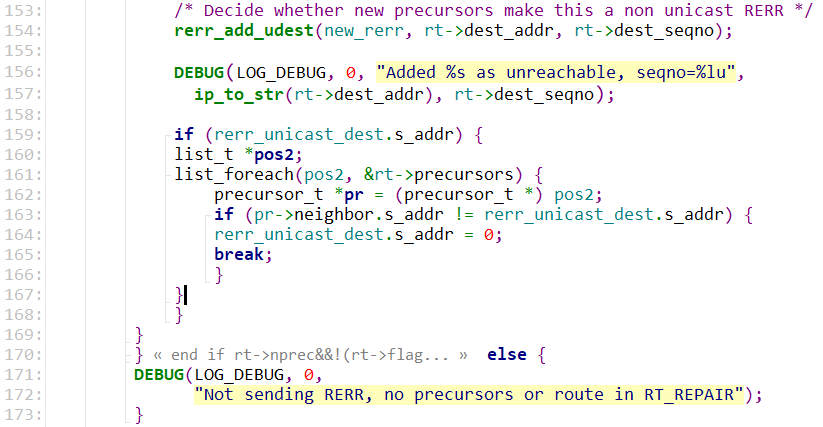
125~126 如果rrer消息未过期，则将通往不可达节点的路由rt置为无效路由

130 更新路由表项里的目的节点序列号

135 如果先驱表不空，路由修复也失败

137~146 新建一个rerr消息，对其各项参数赋值，并输出调试信息

148~150 如果只有一个需要接受rerr消息的邻居节点，则应向邻居节点单播

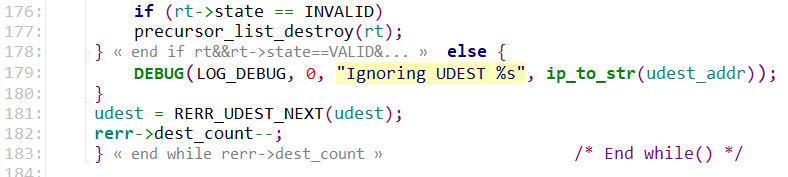


154 添加新的不可达节点

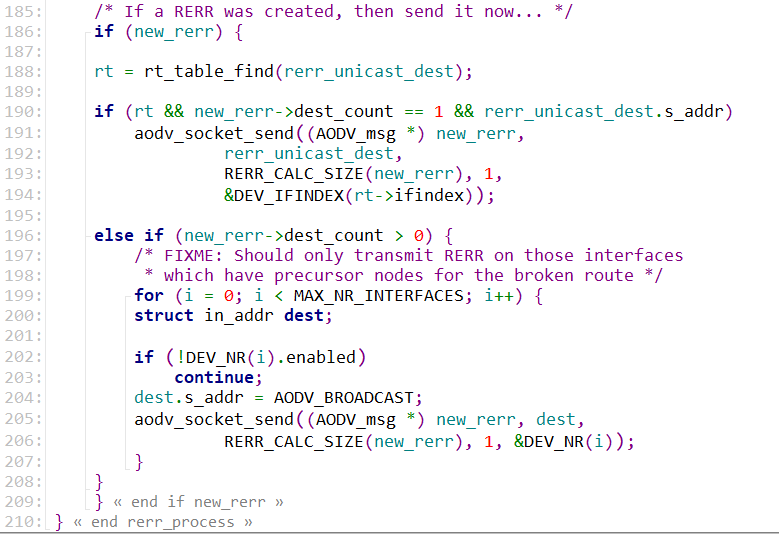
156~157 输出调试信息

159~168 是否有新的先驱节点使其成为非单播的rerr

171~172 输出调试信息



176~182 所有无效路由都将被删除并输出调试信息，处理下一个不可达节点



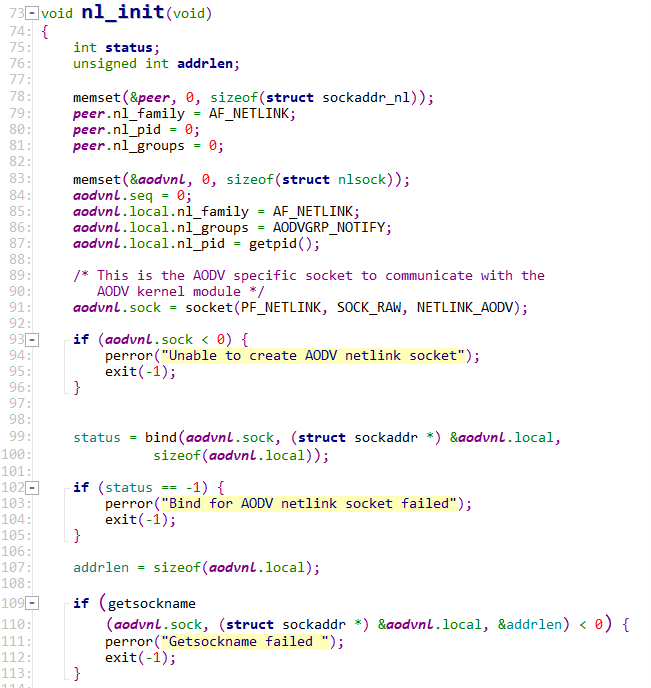
186~194 对于在路由表里能找到有效路由表项的节点，rerr消息直接发出

196~206 没有有效路由且其邻居节点大于0，则从所有可以发送消息的接口把rerr消息广播出去

### 2.2.14 nl.c

|  |  |
| --- | --- |
| [void nl\_init(void)](#_void_nl_init(void)) | 建立与内核模块进行通信以及传递路由消息的套接字 |
| void nl\_cleanup(void) | 关闭套接字 |
| [static void nl\_kaodv\_callback(int sock)](#_static_void_nl_kaodv_callback(int) | 根据从与内核模块进行通信的套接字接收到的消息类型选用合适的回调函数 |
| [static void nl\_rt\_callback(int sock)](#_static_void_nl_rt_callback(int) | 根据从与内核模块传递路由的套接字接收到的消息类型选用合适的回调函数 |
| [int prefix\_length(int family, void \*nm)](#_int_prefix_length(int_family," \o "分析部分) | 根据参数指定的地址族和IP地址计算出子网掩码长度即前缀长度 |
| [int addattr(struct nlmsghdr \*n, int type, void \*data, int alen)](#_int_addattr(struct_nlmsghdr) | 用于更新消息，增加一个属性 |
| [int nl\_send(struct nlsock \*nl, struct nlmsghdr \*n)](#_int_nl_send(struct_nlsock) | 把 netlink 消息嵌入到一个 iovec 结构体中然后用这个结构体构建msghdr消息，并将该消息发送给内核。 |
| [int nl\_kern\_route(int action, int flags, int family,int index, struct in\_addr \*dst, struct in\_addr \*gw,struct in\_addr \*nm, int metric)](#_int_nl_kern_route(int_action,) | 把 netlink 消息嵌入到一个 iovec 结构体中然后用这个结构体构建msghdr消息，并将该消息发送给内核。 |
| [int nl\_send\_add\_route\_msg(struct in\_addr dest, struct in\_addr next\_hop,int metric, u\_int32\_t lifetime, int rt\_flags,int ifindex)](#_int_nl_send_add_route_msg(struct_in) | 用来添加、移除、更新内核路由表中表项的函数 |
| [int nl\_send\_no\_route\_found\_msg(struct in\_addr dest)](#_int_nl_send_no_route_found_msg(stru) | 向内核发送一条netlink消息，将到达dest的路由信息设置为没有该条路由。 |
| [int nl\_send\_del\_route\_msg(struct in\_addr dest, struct in\_addr next\_h](#_int_nl_send_del_route_msg(struct_in) | 向内核发送消息将areq结构体中的路由m按照头部n的指示进行删除，随后调用nl\_kern\_route函数对路由表项进行删除 |
| [int nl\_send\_conf\_msg(void)](#_int_nl_send_conf_msg(void)) | 通过套接字向内核发送类型为配置消息的areq消息。 |

void nl\_init(void):



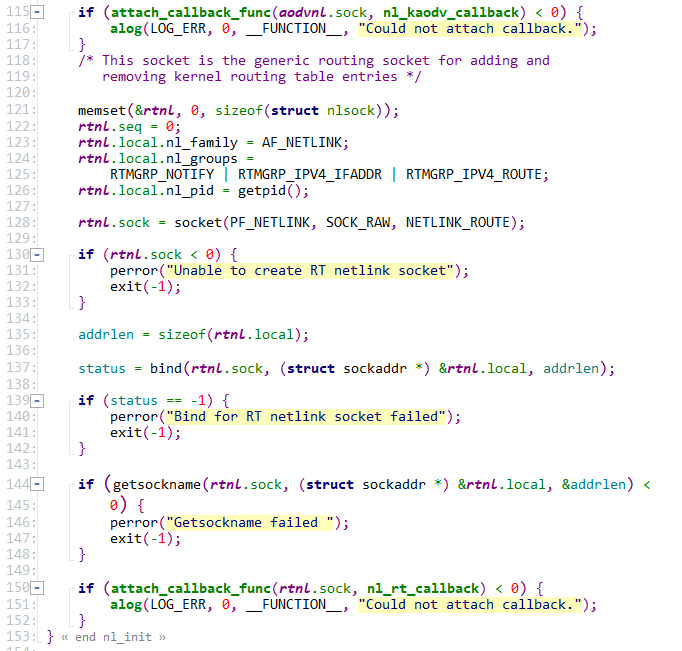
函数的前半部分

78~81 peer是一个sockaddr\_nl结构的实例，进行初始化为0操作后，对其进行各项参数填充，nl\_pid为进程id，nl\_groups为多播组掩码。

83~87 nlsock为自定义的结构体，aodvnl为一个nlsock的实例，同样初始化为0后进行其参数填充。

91 创建AODV专用套接字，指定协议族、套接字类型、使用协议，该套接字用来与内核模块进行通信

99~100 进行套接字的地址绑定

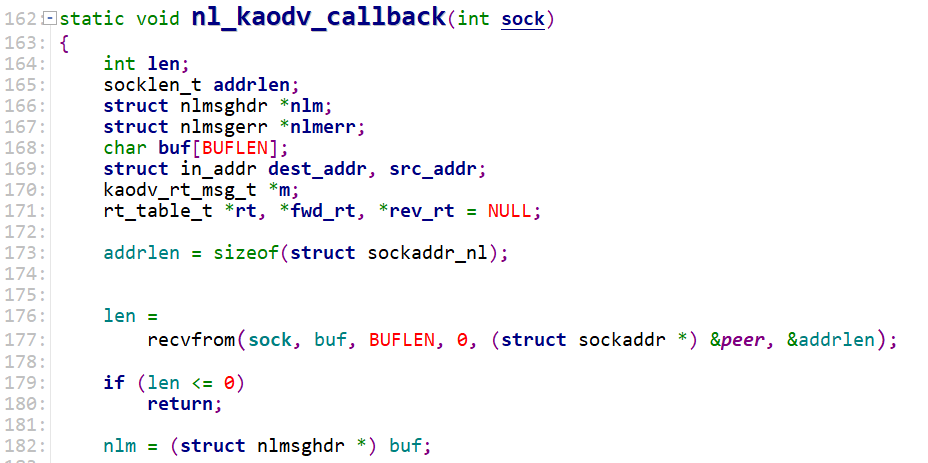


函数的后半部分

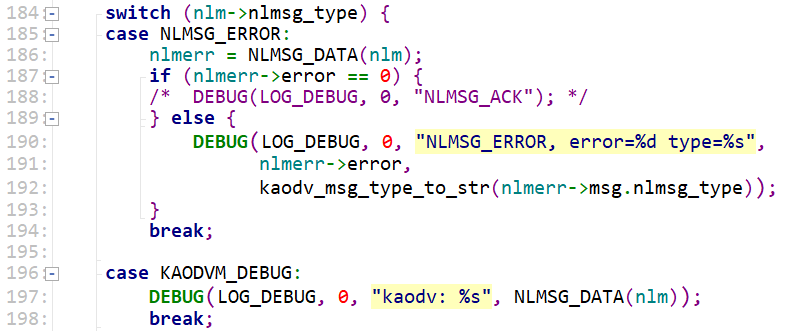
115~117 没有匹配的回调函数

121~152 和前面操作相同，也是新建一个套接字，不过这个套接字是用来与内核传递路由消息的。

static void nl\_kaodv\_callback(int sock):



176~177 从套接字接收内核发来的消息，存到buf中

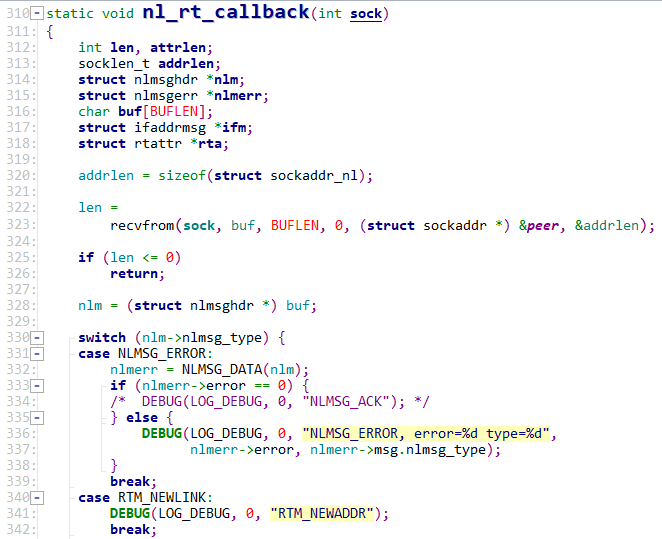


185~194 如果是ERROR消息，打印调试信息

196~198 如果是DEBUG信息，打印调试信息

之后的几个判断，也是同样，如果是TIMEOUT信息，调用route\_expire\_timeot函数，如果是路由请求消息，调用rreq\_route\_discovery函数等。

static void nl\_rt\_callback(int sock):



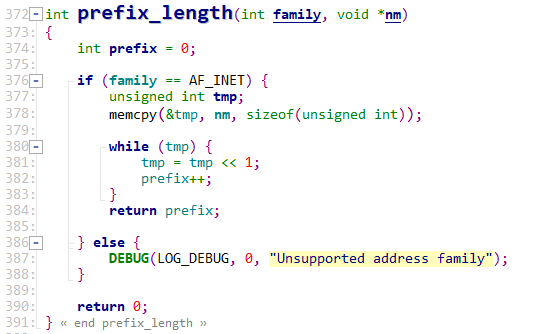
和nl\_kaodv\_callback一样，只是这个函数用来根据与内核进行路由消息传递的套接字的接收内容来决定调用什么函数。

331~339 如果是ERROR消息，打印调试信息

340~342 如果是RTM\_NEWLINK消息打印调试信息

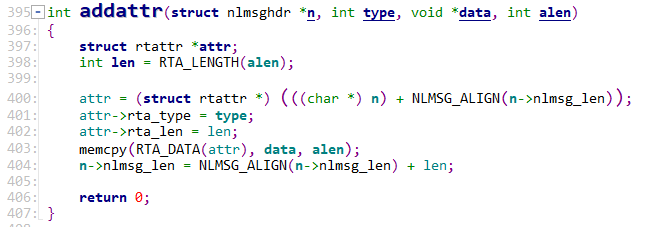
后半部分代码同上，如果是RTM\_NEWADDR、RTM\_NEWROUTE也是打印对应调试信息

int prefix\_length(int family, void \*nm):



根据参数指定的地址族和IP地址计算出子网掩码长度即前缀长度

int addattr(struct nlmsghdr \*n, int type, void \*data, int alen):



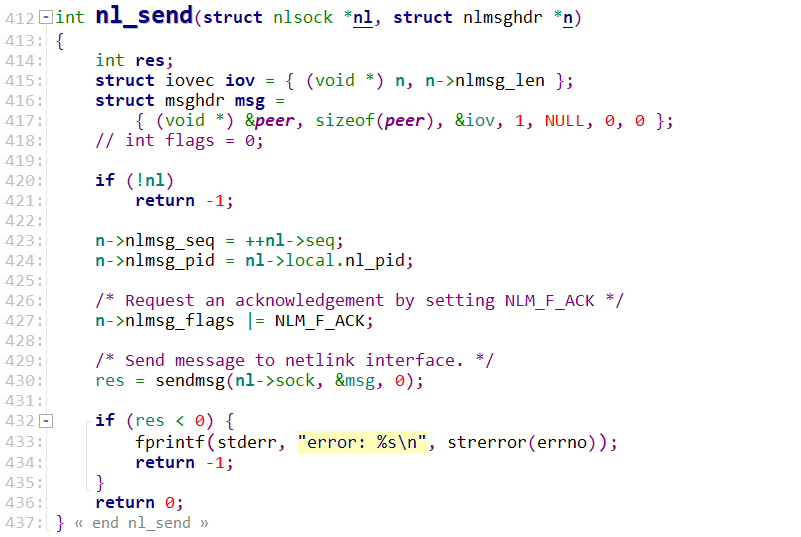
用于更新消息，增加一个属性

397 声明一个消息属性的变量attr

398 原长度更新为加上attr头部长度的总长

400~404 用于对齐和属性初始化。其中宏 NLMSG\_ALIGN (len) 用于得到不小于 len 且字节对齐的最小数值。

int nl\_send(struct nlsock \*nl, struct nlmsghdr \*n):



把 netlink 消息嵌入到一个 iovec 结构体中然后用这个结构体构建msghdr消息，并将该消息发送给内核。

415~417 声明一个msghdr类型的变量，主要用于向socket发送消息或接收消息

int nl\_kern\_route(int action, int flags, int family, int index, struct in\_addr \*dst, struct in\_addr \*gw, struct in\_addr \*nm, int metric):

用来添加、移除、更新内核路由表中表项的函数

int nl\_send\_add\_route\_msg(struct in\_addr dest, struct in\_addr next\_hop,int metric, u\_int32\_t lifetime, int rt\_flags, int ifindex):

向内核发送一条netlink消息，消息内的areq消息头指示内核在表中添加一条路由信息

int nl\_send\_no\_route\_found\_msg(struct in\_addr dest):

向内核发送一条netlink消息，将到达dest的路由信息设置为没有该条路由。

int nl\_send\_del\_route\_msg(struct in\_addr dest, struct in\_addr next\_hop, int metric):

向内核发送消息将areq结构体中的路由m按照头部n的指示进行删除，随后调用nl\_kern\_route函数对路由表项进行删除

int nl\_send\_conf\_msg(void):

通过套接字向内核发送类型为配置消息的areq消息。

### 2.2.15 defh.h

|  |  |
| --- | --- |
| struct dev\_info | 定义一个结构体用于存储设备接口信息 |
| struct host\_info | 定义一个结构体用于存储网络中一个主机的信息 |
| static inline unsigned int ifindex2devindex(unsigned int ifindex) | 根据所给接口号查找在设备数组中的位置 |
| static inline struct dev\_info \*devfromsock(int sock) | 根据所给接口绑定的套接字查找存储该接口的地址 |
| static inline int name2index(char \*name) | 根据名字来查找接口号 |
| struct AODV\_msg | 一个AODV消息的通用结构体 |

### 2.2.16 endian.c

|  |  |  |
| --- | --- | --- |
| 函数名 | 返回值 | 功能 |
| litend(void) | int | 判断主机字节序是不是小端 |
| bigend(void) | int | 判断主机字节序是不是大端 |
| main(int argc, char \*\*argv) | int | 调用判断函数 |