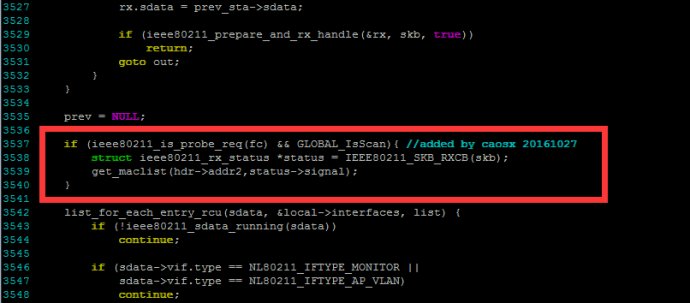
ar71xx上WiFi探针的实现 (probe为内核空间，maclist为用户空间)

1，修改无线驱动mac80211文件夹中的rx.c文件，获取probe帧的mac。

vim build\_dir/target-mips\_34kc\_uClibc-0.9.33.2/linux-ar71xx\_generic/compat-wireless-2016-01-10/net/mac80211/rx.c

找到 static void \_\_ieee80211\_rx\_handle\_packet(struct ieee80211\_hw \*hw,

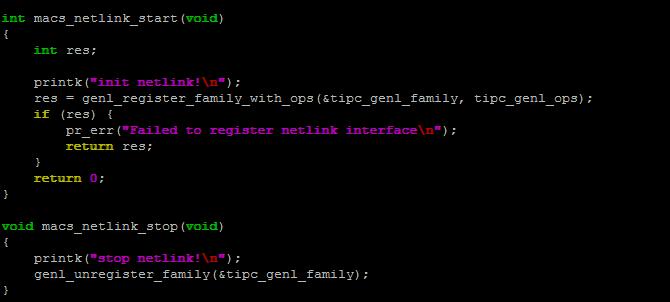
                                         struct sk\_buff \*skb)函数进行修改

[](http://photo.blog.sina.com.cn/showpic.html#blogid=636a55070102wpfx&url=http://album.sina.com.cn/pic/001OSom3zy75XWL1G5kda)  
上图红框中的代码是添加的，struct ieee80211\_rx\_status \*status = IEEE80211\_SKB\_RXCB(skb);用于获取sta设备状态，这里增加这行代码是为了获取status->signal，也就是信号强度。

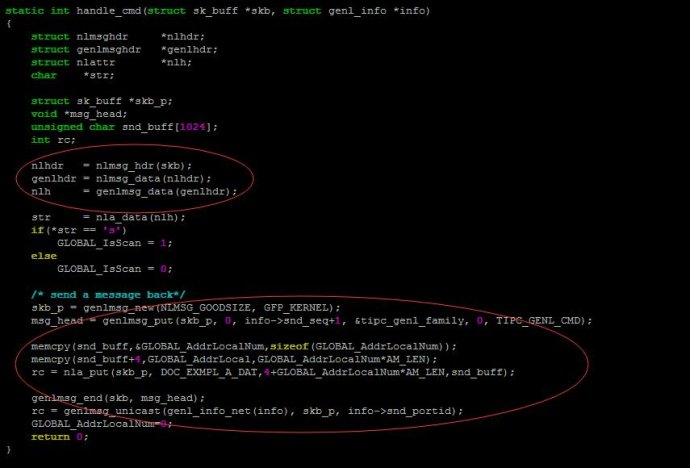
get\_maclist是我自己写的一个函数，用于处理hdr->addr2（probe帧中取得的mac）、status->signal。

2，mac80211文件夹下编写csx\_netlink.c、csx\_netlink.h两个文件，利用genetlink将mac数据由内核层传送到用户层（当然你也可以使用其他方法来实现），以下为主要部分代码：

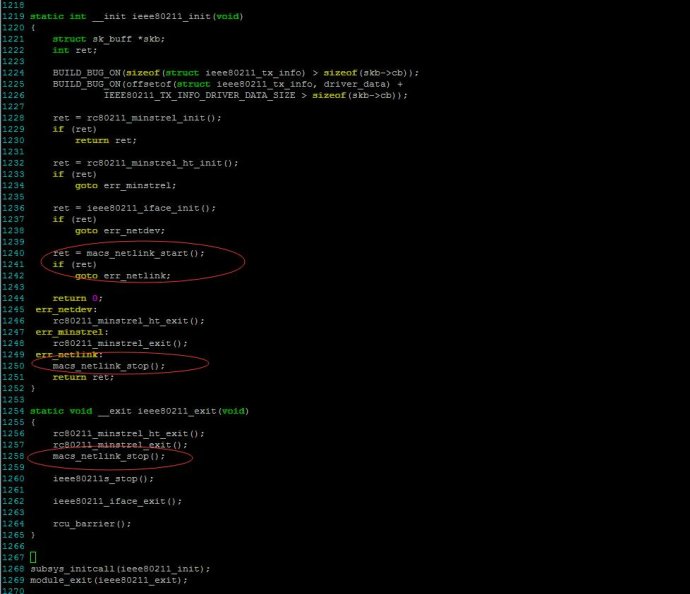
初始化

[](http://photo.blog.sina.com.cn/showpic.html#blogid=636a55070102wpfx&url=http://album.sina.com.cn/pic/001OSom3zy75XXsuUr21c)

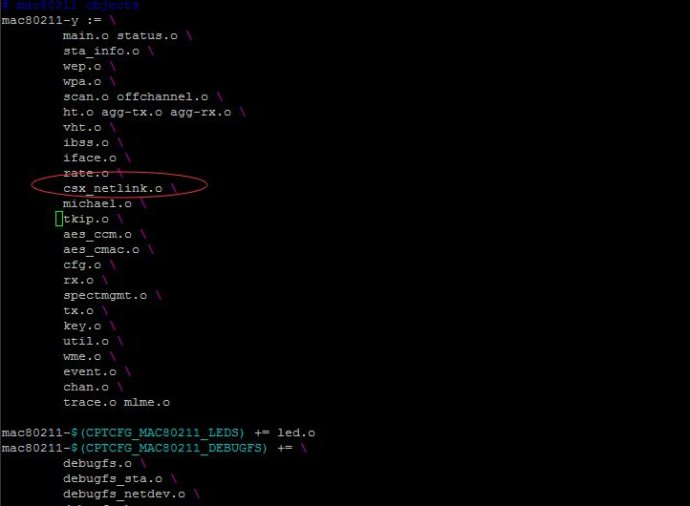
数据接收与发送

[](http://photo.blog.sina.com.cn/showpic.html#blogid=636a55070102wpfx&url=http://album.sina.com.cn/pic/001OSom3zy75XXn790d49)

3，修改mac80211文件夹中的main.c，初始化时加入对csx\_netlink的初始化

[](http://photo.blog.sina.com.cn/showpic.html#blogid=636a55070102wpfx&url=http://album.sina.com.cn/pic/001OSom3zy75XY6woC6ba)

4，修改mac80211文件夹中的Makefile,编译csx\_netlink

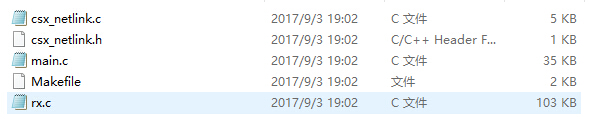
[](http://photo.blog.sina.com.cn/showpic.html#blogid=636a55070102wpfx&url=http://album.sina.com.cn/pic/001OSom3zy75XY0j7tn61)

PS:

1, genetlink的具体实现请参考上一篇文章

2, wifi驱动内核源码下载地址： http://download.csdn.net/download/caoshunxin01/9990463 （需要积分）

压缩包内容 ：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=636a55070102wpfx&url=http://album.sina.com.cn/pic/001OSom3zy7er3gzNh630)

3， 用户空间示例程序：  https://gitee.com/caosx/maclist.git   （开源）

genetlink

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif (2016-10-28 09:03:53)

|  |  |
| --- | --- |
| 标签：  [it](http://search.sina.com.cn/?c=blog&q=it&by=tag) | 分类： [openwrt](http://blog.sina.com.cn/s/articlelist_1667912967_10_1.html) |

转自： http://www.tuicool.com/articles/jE7nim

netlink socket是一种用于用户态进程和内核态进程之间的通信机制。它通过为内核模块提供一组特殊的API，并为用户程序提供了一组标准的socket接口的方式，实现了全双工的通讯连接。

Netlink的特点：

* 双向传输，异步通信
* 用户空间中使用标准socket API
* 内核空间中使用专门的API
* 支持多播
* 可由内核端发起通信
* 支持32种协议类型

netlink仅支持32种协议类型，这在实际应用中可能并不足够。因此产生了generic netlink（以下简称为genl）。   
generic netlink支持1023个子协议号，弥补了netlink协议类型较少的缺陷。支持协议号自动分配。它基于netlink，但是在内核中，generic netlink的接口与netlink并不相同。

**1. Generic Netlink框架概述**

图1表示了Generic Netlink框架。Kernel socket API向用户空间和内核空间分别提供接口。   
Netlink子系统(1)是所有genl通信的基础。Netlink子系统中收到的所有Generic类型的netlink数据都被送到genl总线(2)上；从内核发出的数据也经由genl总线送至netlink子系统，再打包送至用户空间。   
Generic Netlink控制器(4)作为内核的一部分，负责动态地分配genl通道(即genl family id)，并管理genl任务。genl控制器是一个特殊的genl内核用户，它负责监听genl bus上的通信通道。genl通信建立在一系列的通信通道的基础上，每个genl family对应多个通道，这些通道由genl控制器动态分配。

+---------------------+ +---------------------+ | (3) application "A" | | (3) application "B" | +------+--------------+ +--------------+------+ | | \ / \ / | | +-------+--------------------------------+-------+ | : : | user-space =====+ : (5) Kernel socket API : +================ | : : | kernel-space +--------+-------------------------------+-------+ | | +-----+-------------------------------+----+ | (1) Netlink subsystem | +---------------------+--------------------+ | +---------------------+--------------------+ | (2) Generic Netlink bus | +--+--------------------------+-------+----+ | | | +-------+---------+ | | | (4) Controller | / \ +-----------------+ / \ | | +------------------+--+ +--+------------------+ | (3) kernel user "X"| | (3) kernel user "Y" | +---------------------+ +---------------------+

图1：generic netlink框架

**2 Generic Netlink相关结构体   
2.1 genl family**  
Generic Netlink是基于客户端-服务端模型的通信机制。服务端注册family（family是对genl服务的各项定义的集合）。控制器和客户端都通过已注册的信息与服务端通信。   
genl family的结构体如下：

**struct** genl\_family { **unsigned** **int** **id**; **unsigned** **int** hdrsize; **char** name[GENL\_NAMSIZ]; **unsignedint** version; **unsigned** **int** maxattr; **struct** nlattr \*\* attrbuf; **struct** list\_head ops\_list; **struct**list\_head family\_list; };

对此结构体元素具体解释如下：   
  
\* id: family id。当新注册一个family的时候，应该用GENL\_ID\_GENERATE宏(0x0)，表示请控制器自动为family分配的一个id。0x10保留供genl控制器使用。   
\* hdrsize: 用户自定议头部长度。即图2中User Msg的长度。如果没有用户自定义头部，这个值被赋为0。   
\* version: 版本号，一般填1即可。   
\* name: family名，要求不同的family使用不同的名字。以便控制器进行正确的查找。   
\* maxattr：genl使用netlink标准的attr来传输数据。此字段定义了最大attr类型数。（注意：不是一次传输多少个attr，而是一共有多少种attr，因此，这个值可以被设为0，为0代表不区分所收到的数据的attr type）。在接收数据时，可以根据attr type，获得指定的attr type的数据在整体数据中的位置。 

\* struct nlattr \*\*attrbuf   
\* struct list\_head ops\_list   
\* struct list\_head family\_list   
以上的三个字段为私有字段，由系统自动配置，开发者不需要做配置。

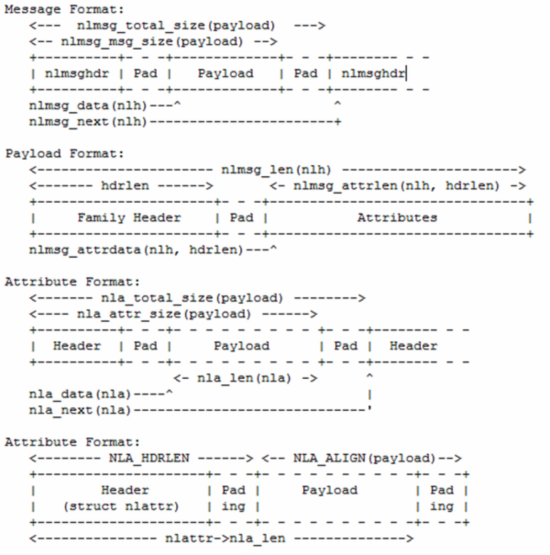


图2 genl报文与linux中各变量的对应关系

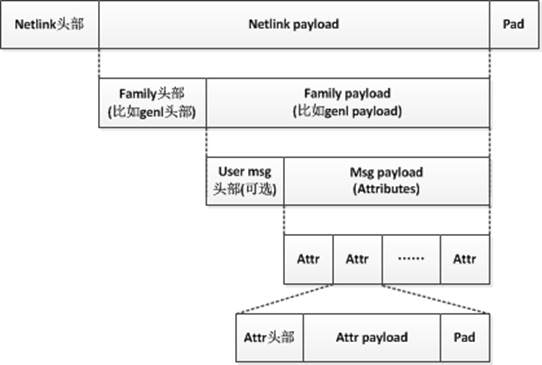


图3 genl报文格式

**2.2 genl\_ops 结构体**

**struct** **genl\_ops** { **u8** cmd; unsigned **int** flags; **struct** **nla\_policy** \*policy; **int** (\*doit)(**structsk\_buff** \*skb, **struct** **genl\_info** \*info); **int** (\*dumpit)(**struct** **sk\_buff** \*skb, **structnetlink\_callback** \*cb); **struct** **list\_head** ops\_list; };

* cmd: 命令名。用于识别各genl\_ops
* flag: 各种设置属性，以“或”连接。在需要admin特权级别时，使用GENL\_ADMIN\_PERM
* policy：定义了attr规则。如果此指针非空，genl在触发事件处理程序之前，会使用这个字段来对帧中的attr做校验（见nlmsg\_parse函数）。该字段可以为空，表示在触发事件处理程序之前，不做校验。

        policy是一个struct nla\_policy的数组。struct nla\_policy结构体表示如下：

**struct** **nla\_policy** { u16 **type**; u16 len; };

        其中，type字段表示attr中的数据类型，可被配置为：   
       NLA\_UNSPEC--未定义   
       NLA\_U8, NLA\_U16, NLA\_U32, NLA\_U64为8bits, 16bits, 32bits, 64bits的无符号整型   
       NLA\_STRING--字符串   
       NLA\_NUL\_STRING--空终止符字符串   
       NLA\_NESTED--attr流   
  
       len字段的意思是：如果在type字段配置的是字符串有关的值，要把len设置为字符串的最大长度（不包含结尾的'\0'）。如果type字段未设置或被设置为NLA\_UNSPEC，那么这里要设置为attr的payload部分的长度。 

* doit：这是一个回调函数。在generic netlink收到数据时触发，运行在进程上下文。

        doit传入两个参数，skb为触发此回调函数的socket buffer。第二个参数是一个genl\_info结构体，定义如下：

**struct** **genl\_info** { **u32** snd\_seq; **u32** snd\_pid; **struct** **nlmsghdr** \* nlhdr; **struct** **genlmsghdr** \* genlhdr; void \* userhdr; **struct** **nlattr** \*\* attrs; };

                 \* snd\_seq：发送序号

                 \* snd\_pid：发送客户端的PID

                 \* nlhdr：netlink header的指针

                 \* genlmsghdr：genl头部的指针（即family头部）

                 \* userhdr：用户自定义头部指针

                \* attrs：attrs，如果定义了genl\_ops->policy，这里的attrs是被policy过滤以后的结果。在完成了操作以后，如果执行正确，返回0；否则，返回一个负数。负数的返回值会触发NLMSG\_ERROR消息。当genl\_ops的flag标志被添加了NLMSG\_ERROR时，即使doit返回0，也会触发NLMSG\_ERROR消息。

* dumpit

这是一个回调函数，当genl\_ops的flag标志被添加了NLM\_F\_DUMP以后，每次收到genl消息即会回触发这个函数。dumpit与doit的区别是:dumpit的第一个参数skb不会携带从客户端发来的数据。相反地，开发者应该在skb中填入需要传给客户端的数据，然后，并skb的数据长度（可以用skb->len）return。skb中携带的数据会被自动送到客户端。只要dumpit的返回值大于0，dumpit函数就会再次被调用，并被要求在skb中填入数据。当服务端没有数据要传给客户端时，dumpit要返回0。如果函数中出错，要求返回一个负值。关于doit和dumpit的触发过程，可以查看源码中的genl\_rcv\_msg函数。

* ops\_list

为私有字段，由系统自动配置，开发者不需要做配置。

**3 Generic Netlink服务端（内核）初始化**

初始化Generic Netlink的过程分为以下四步：定义family，定义operation，注册family，注册operation。下面通过一个简单例子来说明如何完成Generic Netlink的初始化。我们首先创建一个genl\_family结构体的实例。我们在这里定义一个名为"DOC\_EXMPL"的family

**enum** { DOC\_EXMPL\_A\_UNSPEC, DOC\_EXMPL\_A\_MSG, \_\_DOC\_EXMPL\_A\_MAX, }; #define DOC\_EXMPL\_A\_MAX (\_\_DOC\_EXMPL\_A\_MAX - 1) **static** **struct** **genl\_family** doc\_exmpl\_gnl\_family = { .id = GENL\_ID\_GENERATE, .hdrsize = 0, .name = "DOC\_EXMPL", .version = 1, .maxattr = DOC\_EXMPL\_A\_MAX, };

以上，我们定义了一个仅有一种attribuste type的family。.id被配置为GENL\_ID\_GENERATE，指示genl控制器自动分配一个id。   
  
第二步为family创建operations。我们至少要创建一个genl\_ops结构体的实例。 

**int** doc\_exmpl\_echo(**struct** **sk\_buff** \*skb, **struct** **genl\_info** \*info) { } **static** **struct** **nla\_policy**doc\_exmpl\_genl\_policy = [DOC\_EXMPL\_A\_MAX + 1] = { [DOC\_EXMPL\_A\_MSG] = { .**type** = NLA\_NUL\_STRING }, } **enum** { DOC\_EXMPL\_C\_UNSPEC, DOC\_EXMPL\_C\_ECHO, \_\_DOC\_EXMPL\_C\_ECHO, }; #define DOC\_EXMPL\_C\_MAX (\_\_DOC\_EXMPL\_C\_MAX - 1) **struct** **genl\_ops** doc\_exmpl\_gnl\_ops\_echo = { .cmd = DOC\_EXMPL\_C\_ECHO, .flags = 0, .policy = doc\_exmpl\_genl\_policy, .doit = genl\_recv\_doit, .dumpit = NULL, }

这里，我们把attribute policy设为NLA\_NUL\_STRING，表示attr中数据的属性为无NULL结尾的字符串。控制器在收到数据时会自动完成这一类型检查。   
  
我们定义一个operation，它的id为DOC\_EXMPL\_C\_ECHO，把上述的policy配置给它。一旦本family的genl消息在被总到genl总线上，doit函数（doc\_exmpl\_echo）会被调用。   
  
接下来两步是注册family和注册operations。   
  
genl\_register\_family(&doc\_exmpl\_gnl\_family);   
  
genl\_register\_ops(&doc\_exmpl\_gnl\_family, &doc\_exmpl\_gnl\_ops\_echo);   
  
在完成genl操作后，记对完成对family的注销操作。   
  
genl\_unregister\_family(&doc\_exmpl\_gnl\_family);   
  
**4 Generic Netlink客户端（用户空间）初始化** 

Generic Netlink在用户空间的初始化和通常的socket通信一致。大致分为两步，创建socket，把socket绑定到地址上（bind）。   
  
下面也通过一个例子简要说明一下用户空间genl初始化的过程。

**struct** **sockaddr\_nl** saddr; **int** sock; sock = socket(AF\_NETLINK, SOCK\_RAW, NETLINK\_GENERIC); **if**(sock < 0) { **return** -1; } memset(&saddr, 0, **sizeof**(saddr)); saddr.nl\_family = AF\_NETLINK; saddr.nl\_pid = getpid(); **if** (bind(sock, (**struct** **sockaddr**\*)&saddr, **sizeof**(saddr)) < 0) { printf("bind fail!\n"); close(\*p\_sock); **return** -1; }

上述代码中，我们先创建一个socket，注意，第一个参数必须为AF\_NETLINK 或 PF\_NETLINK，表示创建netlink socket，第二个参数必须是SOCK\_RAW或SOCK\_DGRAM， 第三个参数指定netlink协议类型，我们要使用generic netlink，那么就要将其设置为：NETLINK\_GENERIC。   
  
接下来，对于genl不可缺少的一步就是获取family id。family id是服务端注册family时，由控制器自动分配的。此时客户端尚不知道family id为多少，因此需要向客户端请求family id。   
  
下面是一段获取family id的函数 

**static** **int** genl\_get\_family\_id(**int** sd, **char** \*family\_name) { msgtemplate\_t ans; **int** **id**, rc;**struct** nlattr \*na; **int** rep\_len; rc = genl\_send\_msg(sd, GENL\_ID\_CTRL, 0, CTRL\_CMD\_GETFAMILY, 1,CTRL\_ATTR\_FAMILY\_NAME, (**void** \*)family\_name, strlen(family\_name)+1); rep\_len = recv(sd, &ans, **sizeof**(ans), 0); **if** (rep\_len < 0) { **return** 0; } **if** (ans.n.nlmsg\_type == NLMSG\_ERROR || !NLMSG\_OK((&ans.n), rep\_len)) { **return** 0; } na = (**struct** nlattr \*) GENLMSG\_DATA(&ans); na = (**struct** nlattr \*) ((**char** \*) na + NLA\_ALIGN(na->nla\_len)); **if** (na->nla\_type == CTRL\_ATTR\_FAMILY\_ID) { **id** = \*(\_\_u16 \*) NLA\_DATA(na); } **else** { **id** = 0; } **return** **id**; }

在这个函数中，调用genl\_send\_msg（这个函数会在下文中介绍并给出源码）发送请求family id的消息，并调用recv接收服务端的反馈消息。这个消息中即包含了family id。   
  
这个函数的第一个参数是已创建好的socket。第二个参数是family name，注意这里family name需要与服务端注册famile时的name字段一致。该函数返回值即是family id以下是一个调用示例。   
int fid = genl\_get\_family\_id(sock, "DOC\_EXMPL"); 

**5 Generic Netlink通信**  
这一节对如何使用Generic Netlink完成内核空间与用户空间的通信做介绍。并把我的示例代码贡献出来。   
  
示例代码呈现了内核（服务端）和用户空间（客户端）收发数据的过程。   
**5.1 内核发送数据**  
以下是内核端发送数据的源码。在genl\_msg\_send\_to\_user中，调用genl\_msg\_prepare\_usr\_msg和genl\_msg\_mk\_usr\_msg来准备socket buffer，为数据加上各种数据头（参考图2）。genlmsg\_end把整个数据打包完成，通过genlmsg\_unicast完成单播发送。

**int** **genl\_msg\_send\_to\_user**(**void** \*data, **int** len, **pid\_t** pid) { **struct** sk\_buff \*skb; **size\_t** size;**void** \*head; **int** rc; size = nla\_total\_size(len); rc = genl\_msg\_prepare\_usr\_msg(DOC\_EXMPL\_C\_ECHO, size, pid, &skb); **if** (rc) { **return** rc; } rc = genl\_msg\_mk\_usr\_msg(skb, DOC\_EXMPL\_A\_MSG, data, len); **if** (rc) { kfree\_skb(skb); **return** rc; } head = genlmsg\_data(nlmsg\_data(nlmsg\_hdr(skb))); rc = genlmsg\_end(skb, head); **if** (rc < 0) { kfree\_skb(skb); **return** rc; } rc = genlmsg\_unicast(&init\_net, skb, pid); **if** (rc < 0) { **return** rc; } **return** 0; } **static** **inline** **intgenl\_msg\_mk\_usr\_msg**(**struct** sk\_buff \*skb, **int** type, **void** \*data, **int** len) { **int** rc; **if** ((rc = nla\_put(skb, type, len, data)) != 0) { **return** rc; } **return** 0; } **static** **inline** **intgenl\_msg\_prepare\_usr\_msg**(u8 cmd, **size\_t** size, **pid\_t** pid, **struct** sk\_buff \*\*skbp) { **struct**sk\_buff \*skb; skb = genlmsg\_new(size, GFP\_KERNEL); **if** (skb == NULL) { **return** -ENOMEM; } genlmsg\_put(skb, pid, 0, &genl\_family, 0, cmd); \*skbp = skb; **return** 0; }

**5.2 用户空间接收数据**   
客户端调用通用的recv函数即可完成从内核来的数据的接收。需要注意的是，接收到的数据包含几级的header（图3），我们需要准确地定位到我们所需数据的位置。   
  
当没有用户自定义头部（即图3中的User Msg，在注册family时把hdrsize置0）时，可以构建这样的数据结构用于接收数据。这样，收到的数据中的netlink header和genl header就被很容易地剥离开来。 

typedef **struct** **msgtemplate** { **struct** **nlmsghdr** n; **struct** **genlmsghdr** g; **char** data[MAX\_MSG\_SIZE]; } msgtemplate\_t;

下面是客户端接收数据函数的源码： 

#define GENLMSG\_DATA(**glh**) ((**void** \*)(NLMSG\_DATA(glh) + GENL\_HDRLEN)) #define NLA\_DATA(na) ((void \*)((**char** \*)(na) + NLA\_HDRLEN)) void genl\_rcv\_msg(int fid, int sock, char \*\*string) { int ret; struct msgtemplate msg; struct nlattr \*na; ret = recv(**sock**, &msg, sizeof(**msg**), 0); if (**ret** < 0) { return; } //printf("received length %d\n", ret); if (**msg**.n.nlmsg\_type == NLMSG\_ERROR || !NLMSG\_OK((**&msg**.n), ret)) { return; } if (**msg**.n.nlmsg\_type == fid && fid != 0) { na = (**struct**nlattr \*) GENLMSG\_DATA(&msg); \*string = (**char** \*)NLA\_DATA(na); } }

以上函数中，第一个参数为family id，第二个参数为socket，第三个参数为待接收数据的buffer。   
  
**5.3 用户空间发送数据**   
客户端发送数据简单地说就是调用通用的socket API---sendto来发送数据 

**int** **genl\_send\_msg**(**int** sd, **u\_int16\_t** nlmsg\_type, **u\_int32\_t** nlmsg\_pid, **u\_int8\_t** genl\_cmd, **u\_int8\_t** genl\_version, **u\_int16\_t** nla\_type, **void** \*nla\_data, **int** nla\_len) { **struct** nlattr \*na;**struct** sockaddr\_nl nladdr; **int** r, buflen; **char** \*buf; **msgtemplate\_t** msg; **if** (nlmsg\_type == 0) {**return** 0; } msg.n.nlmsg\_len = NLMSG\_LENGTH(GENL\_HDRLEN); msg.n.nlmsg\_type = nlmsg\_type; msg.n.nlmsg\_flags = NLM\_F\_REQUEST; msg.n.nlmsg\_seq = 0; msg.n.nlmsg\_pid = nlmsg\_pid; msg.g.cmd = genl\_cmd; msg.g.version = genl\_version; na = (**struct** nlattr \*) GENLMSG\_DATA(&msg); na->nla\_type = nla\_type; na->nla\_len = nla\_len + 1 + NLA\_HDRLEN; memcpy(NLA\_DATA(na), nla\_data, nla\_len); msg.n.nlmsg\_len += NLMSG\_ALIGN(na->nla\_len); buf = (**char** \*) &msg; buflen = msg.n.nlmsg\_len ;memset(&nladdr, 0, **sizeof**(nladdr)); nladdr.nl\_family = AF\_NETLINK; **while** ((r = sendto(sd, buf, buflen, 0, (**struct** sockaddr \*) &nladdr , **sizeof**(nladdr))) < buflen) { **if** (r > 0) { buf += r; buflen -= r; } **else** **if** (errno != EAGAIN) { **return** -1; } } **return** 0; }

**5.4 内核接收数据**  
内核端一旦收到generic netlink数据，会触发doit函数运行（上文第3节有提及doit的初始化方法）。   
  
doit传入两个参数，skb即是接收到的数据，info包含了Genl消息的一些常用指针。这两个结构体字段详见内核源码。   
  
skb收到的数据还包括了多层的包头，以下程序中的nlmsg\_hdr，nlmsg\_data，genlmsg\_data，nla\_data即是把这些包头层层剥开，para->string指向的数据即是用用户空间传来的“纯数据”。 

**int** genl\_recv\_doit(**struct** **sk\_buff** \*skb, **struct** **genl\_info** \*info) { **static** **int** kthread\_num = 0;**struct** **nlmsghdr** \*nlhdr; **struct** **genlmsghdr** \*genlhdr; **struct** **nlattr** \*nlh; **struct** **thread\_para**\*para; nlhdr = nlmsg\_hdr(skb); genlhdr = nlmsg\_data(nlhdr); nlh = genlmsg\_data(genlhdr); para = (**struct** **thread\_para** \*)kmalloc(**sizeof**(**struct** **thread\_para**), GFP\_KERNEL); para->string = nla\_data(nlh); para->pid = nlhdr->nlmsg\_pid; kthread\_run(thread\_string\_proc, (void \*)(para), "kthread %d", kthread\_num++); **return** 0; }