# EXHIBIT A

```
1/*
2 * INET        An implementation of the TCP/IP protocol suite for the LINUX
3 *             operating system.  INET is implemented using the  BSD Socket
4 *             interface as the means of communication with the user level.
5 *
6 *             ROUTE - implementation of the IP router.
7 *
8 * Version:     @(#)route.c     1.0.14  05/31/93
9 *
10 * Authors:     Ross Biro, <bir7@leland.Stanford.Edu>
11 *             Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12 *             Alan Cox, <gw4pts@gw4pts.ampr.org>
13 *             Linus Torvalds, <Linus.Torvalds@helsinki.fi>
14 *
15 * Fixes:
16 *             Alan Cox        :       Verify area fixes.
17 *             Alan Cox        :       cli() protects routing changes
18 *             Rui Oliveira    :       ICMP routing table updates
19 *             (rco@di.uminho.pt)      Routing table insertion and update
20 *             Linus Torvalds  :       Rewrote bits to be sensible
21 *             Alan Cox        :       Added BSD route gw semantics
22 *             Alan Cox        :       Super /proc >4K
23 *             Alan Cox        :       MTU in route table
24 *             Alan Cox        :       MSS actually. Also added the window
25 *                                     clamper.
26 *             Sam Lantinga    :       Fixed route matching in rt_del()
27 *             Alan Cox        :       Routing cache support.
28 *             Alan Cox        :       Removed compatibility cruft.
29 *             Alan Cox        :       RTF_REJECT support.
30 *             Alan Cox        :       TCP irtt support.
31 *             Jonathan Naylor :       Added Metric support.
32 *     Miquel van Smoorenburg  :       BSD API fixes.
33 *     Miquel van Smoorenburg  :       Metrics.
34 *             Alan Cox        :       Use __u32 properly
35 *             Alan Cox        :       Aligned routing errors more closely with BSD
36 *                                     our system is still very different.
37 *             Alan Cox        :       Faster /proc handling
38 *     Alexey Kuznetsov        :       Massive rework to support tree based routing,
39 *                                     routing caches and better behaviour.
40 *
41 *             Olaf Erb        :       irtt wasn't being copied right.
42 *             Bjorn Ekwall    :       Kerneld route support.
43 *             Alan Cox        :       Multicast fixed (I hope)
44 *             Pavel Krauz     :       Limited broadcast fixed
45 *
46 *             This program is free software; you can redistribute it and/or
47 *             modify it under the terms of the GNU General Public License
48 *             as published by the Free Software Foundation; either version
49 *             2 of the License, or (at your option) any later version.
50 */
51
```

```
52#include <linux/config.h>
53#include <asm/segment.h>
54#include <asm/system.h>
55#include <asm/bitops.h>
56#include <linux/types.h>
57#include <linux/kernel.h>
58#include <linux/sched.h>
59#include <linux/mm.h>
60#include <linux/string.h>
61#include <linux/socket.h>
62#include <linux/sockios.h>
63#include <linux/errno.h>
64#include <linux/in.h>
65#include <linux/inet.h>
66#include <linux/netdevice.h>
67#include <linux/if_arp.h>
68#include <net/ip.h>
69#include <net/protocol.h>
70#include <net/route.h>
71#include <net/tcp.h>
72#include <linux/skbuff.h>
73#include <net/sock.h>
74#include <net/icmp.h>
75#include <net/netlink.h>
76#ifdef CONFIG_KERNELD
77#include <linux/kerneld.h>
78#endif
79
80/*
81 * Forwarding Information Base definitions.
82 */
83
84struct fib_node
85{
86        struct fib_node         *fib_next;
87        __u32                   fib_dst;
88        unsigned long           fib_use;
89        struct fib_info         *fib_info;
90        short                   fib_metric;
91        unsigned char           fib_tos;
92};
93
94/*
95 * This structure contains data shared by many of routes.
96 */
97
98struct fib_info
99{
100        struct fib_info         *fib_next;
101        struct fib_info         *fib_prev;
102        __u32                   fib_gateway;
```

```
103         struct device           *fib_dev;
104         int                     fib_refcnt;
105         unsigned long           fib_window;
106         unsigned short          fib_flags;
107         unsigned short          fib_mtu;
108         unsigned short          fib_irtt;
109};
110
111struct fib_zone
112{
113         struct fib_zone *fz_next;
114         struct fib_node **fz_hash_table;
115         struct fib_node *fz_list;
116         int             fz_nent;
117         int             fz_logmask;
118         __u32           fz_mask;
119};
120
121static struct fib_zone  *fib_zones[33];
122static struct fib_zone  *fib_zone_list;
123static struct fib_node  *fib_loopback = NULL;
124static struct fib_info  *fib_info_list;
125
126/*
127 * Backlogging.
128 */
129
130#define RT_BH_REDIRECT          0
131#define RT_BH_GARBAGE_COLLECT   1
132#define RT_BH_FREE              2
133
134struct rt_req
135{
136         struct rt_req * rtr_next;
137         struct device *dev;
138         __u32 dst;
139         __u32 gw;
140         unsigned char tos;
141};
142
143int                    ip_rt_lock;
144unsigned                ip_rt_bh_mask;
145static struct rt_req    *rt_backlog;
146
147/*
148 * Route cache.
149 */
150
151struct rtable           *ip_rt_hash_table[RT_HASH_DIVISOR];
152static int              rt_cache_size;
153static struct rtable    *rt_free_queue;
```

```
154struct wait_queue       *rt_wait;
155
156static void rt_kick_backlog(void);
157static void rt_cache_add(unsigned hash, struct rtable * rth);
158static void rt_cache_flush(void);
159static void rt_garbage_collect_1(void);
160
161/*
162 * Evaluate mask length.
163 */
164
165static __inline__ int rt_logmask(__u32 mask)
166{
167         if (!(mask = ntohl(mask)))
168                 return 32;
169         return ffz(~mask);
170}
171
172/*
173 * Create mask from length.
174 */
175
176static __inline__ __u32 rt_mask(int logmask)
177{
178         if (logmask >= 32)
179                 return 0;
180         return htonl(~((1<<logmask)-1));
181}
182
183static __inline__ unsigned fz_hash_code(__u32 dst, int logmask)
184{
185         return ip_rt_hash_code(ntohl(dst)>>logmask);
186}
187
188/*
189 * Free FIB node.
190 */
191
192static void fib_free_node(struct fib_node * f)
193{
194         struct fib_info * fi = f->fib_info;
195         if (!--fi->fib_refcnt)
196         {
197#if RT_CACHE_DEBUG >= 2
198                 printk("fib_free_node: fi %08x/%s is free\n", fi->fib_gateway, fi->fib_dev->name);
199#endif
200                 if (fi->fib_next)
201                         fi->fib_next->fib_prev = fi->fib_prev;
202                 if (fi->fib_prev)
203                         fi->fib_prev->fib_next = fi->fib_next;
204                 if (fi == fib_info_list)
```

```
205                     fib_info_list = fi->fib_next;
206             }
207             kfree_s(f, sizeof(struct fib_node));
208}
209
210/*
211 * Find gateway route by address.
212 */
213
214static struct fib_node * fib_lookup_gateway(__u32 dst)
215{
216             struct fib_zone * fz;
217             struct fib_node * f;
218
219             for (fz = fib_zone_list; fz; fz = fz->fz_next)
220             {
221                     if (fz->fz_hash_table)
222                             f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
223                     else
224                             f = fz->fz_list;
225
226                     for ( ; f; f = f->fib_next)
227                     {
228                             if ((dst ^ f->fib_dst) & fz->fz_mask)
229                                     continue;
230                             if (f->fib_info->fib_flags & RTF_GATEWAY)
231                                     return NULL;
232                             return f;
233                     }
234             }
235             return NULL;
236}
237
238/*
239 * Find local route by address.
240 * FIXME: I use "longest match" principle. If destination
241 *       has some non-local route, I'll not search shorter matches.
242 *       It's possible, I'm wrong, but I wanted to prevent following
243 *       situation:
244 *      route add 193.233.7.128 netmask 255.255.255.192 gw xxxxxx
245 *      route add 193.233.7.0   netmask 255.255.255.0 eth1
246 *       (Two ethernets connected by serial line, one is small and other is large)
247 *       Host 193.233.7.129 is locally unreachable,
248 *       but old (<=1.3.37) code will send packets destined for it to eth1.
249 *
250 */
251
252static struct fib_node * fib_lookup_local(__u32 dst)
253{
254             struct fib_zone * fz;
255             struct fib_node * f;
```

```
256
257         for (fz = fib_zone_list; fz; fz = fz->fz_next)
258         {
259                 int longest_match_found = 0;
260
261                 if (fz->fz_hash_table)
262                         f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
263                 else
264                         f = fz->fz_list;
265
266                 for ( ; f; f = f->fib_next)
267                 {
268                         if ((dst ^ f->fib_dst) & fz->fz_mask)
269                                 continue;
270                         if (!(f->fib_info->fib_flags & RTF_GATEWAY))
271                                 return f;
272                         longest_match_found = 1;
273                 }
274                 if (longest_match_found)
275                         return NULL;
276         }
277         return NULL;
278}
279
280/*
281 * Main lookup routine.
282 *      IMPORTANT NOTE: this algorithm has small difference from <=1.3.37 visible
283 *      by user. It doesn't route non-CIDR broadcasts by default.
284 *
285 *      F.e.
286 *              ifconfig eth0 193.233.7.65 netmask 255.255.255.192 broadcast 193.233.7.255
287 *      is valid, but if you really are not able (not allowed, do not want) to
288 *      use CIDR compliant broadcast 193.233.7.127, you should add host route:
289 *              route add -host 193.233.7.255 eth0
290 */
291
292static struct fib_node * fib_lookup(__u32 dst)
293{
294         struct fib_zone * fz;
295         struct fib_node * f;
296
297         for (fz = fib_zone_list; fz; fz = fz->fz_next)
298         {
299                 if (fz->fz_hash_table)
300                         f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
301                 else
302                         f = fz->fz_list;
303
304                 for ( ; f; f = f->fib_next)
305                 {
306                         if ((dst ^ f->fib_dst) & fz->fz_mask)
```

```
307                             continue;
308                     return f;
309             }
310     }
311     return NULL;
312}
313
314static __inline__ struct device * get_gw_dev(__u32 gw)
315{
316     struct fib_node * f;
317     f = fib_lookup_gateway(gw);
318     if (f)
319             return f->fib_info->fib_dev;
320     return NULL;
321}
322
323/*
324 *      Check if a mask is acceptable.
325 */
326
327static inline int bad_mask(__u32 mask, __u32 addr)
328{
329     if (addr & (mask = ~mask))
330             return 1;
331     mask = ntohl(mask);
332     if (mask & (mask+1))
333             return 1;
334     return 0;
335}
336
337
338static int fib_del_list(struct fib_node **fp, __u32 dst,
339             struct device * dev, __u32 gtw, short flags, short metric, __u32 mask)
340{
341     struct fib_node *f;
342     int found=0;
343
344     while((f = *fp) != NULL)
345     {
346             struct fib_info * fi = f->fib_info;
347
348             /*
349              *      Make sure the destination and netmask match.
350              *      metric, gateway and device are also checked
351              *      if they were specified.
352              */
353             if (f->fib_dst != dst ||
354                 (gtw && fi->fib_gateway != gtw) ||
355                 (metric >= 0 && f->fib_metric != metric) ||
356                 (dev && fi->fib_dev != dev) )
357             {
```

```
358                     fp = &f->fib_next;
359                     continue;
360             }
361             cli();
362             *fp = f->fib_next;
363             if (fib_loopback == f)
364                     fib_loopback = NULL;
365             sti();
366             ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric, fi->fib_dev->name);
367             fib_free_node(f);
368             found++;
369     }
370     return found;
371}
372
373static __inline__ int fib_del_1(__u32 dst, __u32 mask,
374             struct device * dev, __u32 gtw, short flags, short metric)
375{
376     struct fib_node **fp;
377     struct fib_zone *fz;
378     int found=0;
379
380     if (!mask)
381     {
382             for (fz=fib_zone_list; fz; fz = fz->fz_next)
383             {
384                     int tmp;
385                     if (fz->fz_hash_table)
386                             fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
387                     else
388                             fp = &fz->fz_list;
389
390                     tmp = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
391                     fz->fz_nent -= tmp;
392                     found += tmp;
393             }
394     }
395     else
396     {
397             if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
398             {
399                     if (fz->fz_hash_table)
400                             fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
401                     else
402                             fp = &fz->fz_list;
403
404                     found = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
405                     fz->fz_nent -= found;
406             }
407     }
408
```

```
409         if (found)
410         {
411                 rt_cache_flush();
412                 return 0;
413         }
414         return -ESRCH;
415}
416
417
418static struct fib_info * fib_create_info(__u32 gw, struct device * dev,
419                                         unsigned short flags, unsigned short mss,
420                                         unsigned long window, unsigned short irtt)
421{
422         struct fib_info * fi;
423
424         if (!(flags & RTF_MSS))
425         {
426                 mss = dev->mtu;
427#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
428                 /*
429                  *      If MTU was not specified, use default.
430                  *      If you want to increase MTU for some net (local subnet)
431                  *      use "route add .... mss xxx".
432                  *
433                  *      The MTU isn't currently always used and computed as it
434                  *      should be as far as I can tell. [Still verifying this is right]
435                  */
436                 if ((flags & RTF_GATEWAY) && mss > 576)
437                         mss = 576;
438#endif
439         }
440         if (!(flags & RTF_WINDOW))
441                 window = 0;
442         if (!(flags & RTF_IRTT))
443                 irtt = 0;
444
445         for (fi=fib_info_list; fi; fi = fi->fib_next)
446         {
447                 if (fi->fib_gateway != gw ||
448                     fi->fib_dev != dev  ||
449                     fi->fib_flags != flags ||
450                     fi->fib_mtu != mss ||
451                     fi->fib_window != window ||
452                     fi->fib_irtt != irtt)
453                         continue;
454                 fi->fib_refcnt++;
455#if RT_CACHE_DEBUG >= 2
456                 printk("fib_create_info: fi %08x/%s is duplicate\n", fi->fib_gateway, fi->fib_dev->name);
457#endif
458                 return fi;
459         }
```

```
460         fi = (struct fib_info*)kmalloc(sizeof(struct fib_info), GFP_KERNEL);
461         if (!fi)
462                 return NULL;
463         memset(fi, 0, sizeof(struct fib_info));
464         fi->fib_flags = flags;
465         fi->fib_dev = dev;
466         fi->fib_gateway = gw;
467         fi->fib_mtu = mss;
468         fi->fib_window = window;
469         fi->fib_refcnt++;
470         fi->fib_next = fib_info_list;
471         fi->fib_prev = NULL;
472         fi->fib_irtt = irtt;
473         if (fib_info_list)
474                 fib_info_list->fib_prev = fi;
475         fib_info_list = fi;
476#if RT_CACHE_DEBUG >= 2
477         printk("fib_create_info: fi %08x/%s is created\n", fi->fib_gateway, fi->fib_dev->name);
478#endif
479         return fi;
480}
481
482
483static __inline__ void fib_add_1(short flags, __u32 dst, __u32 mask,
484         __u32 gw, struct device *dev, unsigned short mss,
485         unsigned long window, unsigned short irtt, short metric)
486{
487         struct fib_node *f, *f1;
488         struct fib_node **fp;
489         struct fib_node **dup_fp = NULL;
490         struct fib_zone * fz;
491         struct fib_info * fi;
492         int logmask;
493
494         /*
495          *      Allocate an entry and fill it in.
496          */
497
498         f = (struct fib_node *) kmalloc(sizeof(struct fib_node), GFP_KERNEL);
499         if (f == NULL)
500                 return;
501
502         memset(f, 0, sizeof(struct fib_node));
503         f->fib_dst = dst;
504         f->fib_metric = metric;
505         f->fib_tos    = 0;
506
507         if  ((fi = fib_create_info(gw, dev, flags, mss, window, irtt)) == NULL)
508         {
509                 kfree_s(f, sizeof(struct fib_node));
510                 return;
```

```
511         }
512         f->fib_info = fi;
513
514         logmask = rt_logmask(mask);
515         fz = fib_zones[logmask];
516
517
518         if (!fz)
519         {
520                 int i;
521                 fz = kmalloc(sizeof(struct fib_zone), GFP_KERNEL);
522                 if (!fz)
523                 {
524                         fib_free_node(f);
525                         return;
526                 }
527                 memset(fz, 0, sizeof(struct fib_zone));
528                 fz->fz_logmask = logmask;
529                 fz->fz_mask = mask;
530                 for (i=logmask-1; i>=0; i--)
531                         if (fib_zones[i])
532                                 break;
533                 cli();
534                 if (i<0)
535                 {
536                         fz->fz_next = fib_zone_list;
537                         fib_zone_list = fz;
538                 }
539                 else
540                 {
541                         fz->fz_next = fib_zones[i]->fz_next;
542                         fib_zones[i]->fz_next = fz;
543                 }
544                 fib_zones[logmask] = fz;
545                 sti();
546         }
547
548         /*
549          * If zone overgrows RTZ_HASHING_LIMIT, create hash table.
550          */
551
552         if (fz->fz_nent >= RTZ_HASHING_LIMIT && !fz->fz_hash_table && logmask<32)
553         {
554                 struct fib_node ** ht;
555#if RT_CACHE_DEBUG >= 2
556                 printk("fib_add_1: hashing for zone %d started\n", logmask);
557#endif
558                 ht = kmalloc(RTZ_HASH_DIVISOR*sizeof(struct rtable*), GFP_KERNEL);
559
560                 if (ht)
561                 {
```

```
562                 memset(ht, 0, RTZ_HASH_DIVISOR*sizeof(struct fib_node*));
563                 cli();
564                 f1 = fz->fz_list;
565                 while (f1)
566                 {
567                         struct fib_node * next;
568                         unsigned hash = fz_hash_code(f1->fib_dst, logmask);
569                         next = f1->fib_next;
570                         f1->fib_next = ht[hash];
571                         ht[hash] = f1;
572                         f1 = next;
573                 }
574                 fz->fz_list = NULL;
575                 fz->fz_hash_table = ht;
576                 sti();
577         }
578 }
579
580 if (fz->fz_hash_table)
581         fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
582 else
583         fp = &fz->fz_list;
584
585 /*
586  * Scan list to find the first route with the same destination
587  */
588 while ((f1 = *fp) != NULL)
589 {
590         if (f1->fib_dst == dst)
591                 break;
592         fp = &f1->fib_next;
593 }
594
595 /*
596  * Find route with the same destination and less (or equal) metric.
597  */
598 while ((f1 = *fp) != NULL && f1->fib_dst == dst)
599 {
600         if (f1->fib_metric >= metric)
601                 break;
602         /*
603          *      Record route with the same destination and gateway,
604          *      but less metric. We'll delete it
605          *      after instantiation of new route.
606          */
607         if (f1->fib_info->fib_gateway == gw &&
608            (gw || f1->fib_info->fib_dev == dev))
609                 dup_fp = fp;
610         fp = &f1->fib_next;
611 }
612
```

```
613     /*
614      * Is it already present?
615      */
616
617     if (f1 && f1->fib_metric == metric && f1->fib_info == fi)
618     {
619             fib_free_node(f);
620             return;
621     }
622
623     /*
624      * Insert new entry to the list.
625      */
626
627     cli();
628     f->fib_next = f1;
629     *fp = f;
630     if (!fib_loopback && (fi->fib_dev->flags & IFF_LOOPBACK))
631             fib_loopback = f;
632     sti();
633     fz->fz_nent++;
634     ip_netlink_msg(RTMSG_NEWROUTE, dst, gw, mask, flags, metric, fi->fib_dev->name);
635
636     /*
637      *      Delete route with the same destination and gateway.
638      *      Note that we should have at most one such route.
639      */
640     if (dup_fp)
641             fp = dup_fp;
642     else
643             fp = &f->fib_next;
644
645     while ((f1 = *fp) != NULL && f1->fib_dst == dst)
646     {
647             if (f1->fib_info->fib_gateway == gw &&
648                 (gw || f1->fib_info->fib_dev == dev))
649             {
650                     cli();
651                     *fp = f1->fib_next;
652                     if (fib_loopback == f1)
653                             fib_loopback = NULL;
654                     sti();
655                     ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags, metric, f1->fib_info-
>fib_dev->name);
656                     fib_free_node(f1);
657                     fz->fz_nent--;
658                     break;
659             }
660             fp = &f1->fib_next;
661     }
662     rt_cache_flush();
```

```
663         return;
664}
665
666static int rt_flush_list(struct fib_node ** fp, struct device *dev)
667{
668         int found = 0;
669         struct fib_node *f;
670
671         while ((f = *fp) != NULL) {
672/*
673 *       "Magic" device route is allowed to point to loopback,
674 *       discard it too.
675 */
676                 if (f->fib_info->fib_dev != dev &&
677                     (f->fib_info->fib_dev != &loopback_dev || f->fib_dst != dev->pa_addr)) {
678                         fp = &f->fib_next;
679                         continue;
680                 }
681                 cli();
682                 *fp = f->fib_next;
683                 if (fib_loopback == f)
684                         fib_loopback = NULL;
685                 sti();
686                 fib_free_node(f);
687                 found++;
688         }
689         return found;
690}
691
692static __inline__ void fib_flush_1(struct device *dev)
693{
694         struct fib_zone *fz;
695         int found = 0;
696
697         for (fz = fib_zone_list; fz; fz = fz->fz_next)
698         {
699                 if (fz->fz_hash_table)
700                 {
701                         int i;
702                         int tmp = 0;
703                         for (i=0; i<RTZ_HASH_DIVISOR; i++)
704                                 tmp += rt_flush_list(&fz->fz_hash_table[i], dev);
705                         fz->fz_nent -= tmp;
706                         found += tmp;
707                 }
708                 else
709                 {
710                         int tmp;
711                         tmp = rt_flush_list(&fz->fz_list, dev);
712                         fz->fz_nent -= tmp;
713                         found += tmp;
```

```
714             }
715         }
716
717         if (found)
718             rt_cache_flush();
719}
720
721
722/*
723 *       Called from the PROCfs module. This outputs /proc/net/route.
724 *
725 *       We preserve the old format but pad the buffers out. This means that
726 *       we can spin over the other entries as we read them. Remember the
727 *       gated BGP4 code could need to read 60,000+ routes on occasion (that's
728 *       about 7Mb of data). To do that ok we will need to also cache the
729 *       last route we got to (reads will generally be following on from
730 *       one another without gaps).
731 */
732
733int rt_get_info(char *buffer, char **start, off_t offset, int length, int dummy)
734{
735         struct fib_zone *fz;
736         struct fib_node *f;
737         int len=0;
738         off_t pos=0;
739         char temp[129];
740         int i;
741
742         pos = 128;
743
744         if (offset<128)
745         {
746             sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\tIRTT");
747             len = 128;
748         }
749
750         while  (ip_rt_lock)
751             sleep_on(&rt_wait);
752         ip_rt_fast_lock();
753
754         for (fz=fib_zone_list; fz; fz = fz->fz_next)
755         {
756             int maxslot;
757             struct fib_node ** fp;
758
759             if (fz->fz_nent == 0)
760                 continue;
761
762             if (pos + 128*fz->fz_nent <= offset)
763             {
```

```
764                     pos += 128*fz->fz_nent;
765                     len = 0;
766                     continue;
767             }
768
769             if (fz->fz_hash_table)
770             {
771                     maxslot = RTZ_HASH_DIVISOR;
772                     fp      = fz->fz_hash_table;
773             }
774             else
775             {
776                     maxslot = 1;
777                     fp      = &fz->fz_list;
778             }
779
780             for (i=0; i < maxslot; i++, fp++)
781             {
782
783                     for (f = *fp; f; f = f->fib_next)
784                     {
785                             struct fib_info * fi;
786                             /*
787                              *      Spin through entries until we are ready
788                              */
789                             pos += 128;
790
791                             if (pos <= offset)
792                             {
793                                     len=0;
794                                     continue;
795                             }
796
797                             fi = f->fib_info;
798                             sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%d\t%lu\t%d\t%08lX\t%d\t%lu\t%u",
799                                     fi->fib_dev->name, (unsigned long)f->fib_dst, (unsigned long)fi-
>fib_gateway,
800                                     fi->fib_flags, 0, f->fib_use, f->fib_metric,
801                                     (unsigned long)fz->fz_mask, (int)fi->fib_mtu, fi->fib_window, (int)fi-
>fib_irtt);
802                             sprintf(buffer+len,"%-127s\n",temp);
803
804                             len += 128;
805                             if (pos >= offset+length)
806                                     goto done;
807                     }
808             }
809     }
810
811done:
```

```
812         ip_rt_unlock();
813         wake_up(&rt_wait);
814
815         *start = buffer+len-(pos-offset);
816         len = pos - offset;
817         if (len>length)
818                 len = length;
819         return len;
820}
821
822int rt_cache_get_info(char *buffer, char **start, off_t offset, int length, int dummy)
823{
824         int len=0;
825         off_t pos=0;
826         char temp[129];
827         struct rtable *r;
828         int i;
829
830         pos = 128;
831
832         if (offset<128)
833         {
834                 sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tSource\t\tMTU\tWindow\tIRTT\tHH\tARP");
835                 len = 128;
836         }
837
838
839         while  (ip_rt_lock)
840                 sleep_on(&rt_wait);
841         ip_rt_fast_lock();
842
843         for (i = 0; i<RT_HASH_DIVISOR; i++)
844         {
845                 for (r = ip_rt_hash_table[i]; r; r = r->rt_next)
846                 {
847                         /*
848                          *      Spin through entries until we are ready
849                          */
850                         pos += 128;
851
852                         if (pos <= offset)
853                         {
854                                 len = 0;
855                                 continue;
856                         }
857
858                         sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%d\t%u\t%d\t%08lX\t%d\t%lu\t%u\t%d\t%1d",
859                                 r->rt_dev->name, (unsigned long)r->rt_dst, (unsigned long)r->rt_gateway,
860                                 r->rt_flags, r->rt_refcnt, r->rt_use, 0,
```

```
861                         (unsigned long)r->rt_src, (int)r->rt_mtu, r->rt_window, (int)r->rt_irtt, r->rt_hh ?
r->rt_hh->hh_refcnt : -1, r->rt_hh ? r->rt_hh->hh_uptodate : 0);
862                 sprintf(buffer+len,"%-127s\n",temp);
863                 len += 128;
864                 if (pos >= offset+length)
865                         goto done;
866         }
867 }
868
869done:
870 ip_rt_unlock();
871 wake_up(&rt_wait);
872
873 *start = buffer+len-(pos-offset);
874 len = pos-offset;
875 if (len>length)
876         len = length;
877 return len;
878}
879
880
881static void rt_free(struct rtable * rt)
882{
883 unsigned long flags;
884
885 save_flags(flags);
886 cli();
887 if (!rt->rt_refcnt)
888 {
889         struct hh_cache * hh = rt->rt_hh;
890         rt->rt_hh = NULL;
891         restore_flags(flags);
892         if (hh && atomic_dec_and_test(&hh->hh_refcnt))
893                 kfree_s(hh, sizeof(struct hh_cache));
894         kfree_s(rt, sizeof(struct rt_table));
895         return;
896 }
897 rt->rt_next = rt_free_queue;
898 rt->rt_flags &= ~RTF_UP;
899 rt_free_queue = rt;
900 ip_rt_bh_mask |= RT_BH_FREE;
901#if RT_CACHE_DEBUG >= 2
902 printk("rt_free: %08x\n", rt->rt_dst);
903#endif
904 restore_flags(flags);
905}
906
907/*
908 * RT "bottom half" handlers. Called with masked interrupts.
909 */
910
```

```
911static __inline__ void rt_kick_free_queue(void)
912{
913	struct rtable *rt, **rtp;
914
915	rtp = &rt_free_queue;
916
917	while ((rt = *rtp) != NULL)
918	{
919		if  (!rt->rt_refcnt)
920		{
921			struct hh_cache * hh = rt->rt_hh;
922#if RT_CACHE_DEBUG >= 2
923			__u32 daddr = rt->rt_dst;
924#endif
925			*rtp = rt->rt_next;
926			rt->rt_hh = NULL;
927			sti();
928			if (hh && atomic_dec_and_test(&hh->hh_refcnt))
929				kfree_s(hh, sizeof(struct hh_cache));
930			kfree_s(rt, sizeof(struct rt_table));
931#if RT_CACHE_DEBUG >= 2
932			printk("rt_kick_free_queue: %08x is free\n", daddr);
933#endif
934			cli();
935			continue;
936		}
937		rtp = &rt->rt_next;
938	}
939}
940
941void ip_rt_run_bh()
942{
943	unsigned long flags;
944	save_flags(flags);
945	cli();
946	if (ip_rt_bh_mask && !ip_rt_lock)
947	{
948		if (ip_rt_bh_mask & RT_BH_REDIRECT)
949			rt_kick_backlog();
950
951		if (ip_rt_bh_mask & RT_BH_GARBAGE_COLLECT)
952		{
953			ip_rt_fast_lock();
954			ip_rt_bh_mask &= ~RT_BH_GARBAGE_COLLECT;
955			sti();
956			rt_garbage_collect_1();
957			cli();
958			ip_rt_fast_unlock();
959		}
960
961		if (ip_rt_bh_mask & RT_BH_FREE)
```

```
962                     rt_kick_free_queue();
963         }
964         restore_flags(flags);
965}
966
967
968void ip_rt_check_expire()
969{
970         ip_rt_fast_lock();
971         if (ip_rt_lock == 1)
972         {
973                 int i;
974                 struct rtable *rth, **rthp;
975                 unsigned long flags;
976                 unsigned long now = jiffies;
977
978                 save_flags(flags);
979                 for (i=0; i<RT_HASH_DIVISOR; i++)
980                 {
981                         rthp = &ip_rt_hash_table[i];
982
983                         while ((rth = *rthp) != NULL)
984                         {
985                                 struct rtable * rth_next = rth->rt_next;
986
987                                 /*
988                                  * Cleanup aged off entries.
989                                  */
990
991                                 cli();
992                                 if (!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT < now)
993                                 {
994                                         *rthp = rth_next;
995                                         sti();
996                                         rt_cache_size--;
997#if RT_CACHE_DEBUG >= 2
998                                         printk("rt_check_expire clean %02x@%08x\n", i, rth->rt_dst);
999#endif
1000                                         rt_free(rth);
1001                                         continue;
1002                                 }
1003                                 sti();
1004
1005                                 if (!rth_next)
1006                                         break;
1007
1008                                 /*
1009                                  * LRU ordering.
1010                                  */
1011
```

```
1012                        if (rth->rt_lastuse + RT_CACHE_BUBBLE_THRESHOLD < rth_next-
>rt_lastuse ||
1013                           (rth->rt_lastuse < rth_next->rt_lastuse &&
1014                            rth->rt_use < rth_next->rt_use))
1015                        {
1016#if RT_CACHE_DEBUG >= 2
1017                                printk("rt_check_expire bubbled %02x@%08x<->%08x\n", i, rth->rt_dst,
rth_next->rt_dst);
1018#endif
1019                                cli();
1020                                *rthp = rth_next;
1021                                rth->rt_next = rth_next->rt_next;
1022                                rth_next->rt_next = rth;
1023                                sti();
1024                                rthp = &rth_next->rt_next;
1025                                continue;
1026                        }
1027                        rthp = &rth->rt_next;
1028                }
1029            }
1030            restore_flags(flags);
1031            rt_kick_free_queue();
1032        }
1033        ip_rt_unlock();
1034}
1035
1036static void rt_redirect_1(__u32 dst, __u32 gw, struct device *dev)
1037{
1038        struct rtable *rt;
1039        unsigned long hash = ip_rt_hash_code(dst);
1040
1041        if (gw == dev->pa_addr)
1042                return;
1043        if (dev != get_gw_dev(gw))
1044                return;
1045        rt = (struct rtable *) kmalloc(sizeof(struct rtable), GFP_ATOMIC);
1046        if (rt == NULL)
1047                return;
1048        memset(rt, 0, sizeof(struct rtable));
1049        rt->rt_flags = RTF_DYNAMIC | RTF_MODIFIED | RTF_HOST | RTF_GATEWAY |
RTF_UP;
1050        rt->rt_dst = dst;
1051        rt->rt_dev = dev;
1052        rt->rt_gateway = gw;
1053        rt->rt_src = dev->pa_addr;
1054        rt->rt_mtu = dev->mtu;
1055#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
1056        if (dev->mtu > 576)
1057                rt->rt_mtu = 576;
1058#endif
1059        rt->rt_lastuse  = jiffies;
```

```
1060        rt->rt_refcnt  = 1;
1061        rt_cache_add(hash, rt);
1062        ip_rt_put(rt);
1063        return;
1064}
1065
1066static void rt_cache_flush(void)
1067{
1068        int i;
1069        struct rtable * rth, * next;
1070
1071        for (i=0; i<RT_HASH_DIVISOR; i++)
1072        {
1073                int nr=0;
1074
1075                cli();
1076                if (!(rth = ip_rt_hash_table[i]))
1077                {
1078                        sti();
1079                        continue;
1080                }
1081
1082                ip_rt_hash_table[i] = NULL;
1083                sti();
1084
1085                for (; rth; rth=next)
1086                {
1087                        next = rth->rt_next;
1088                        rt_cache_size--;
1089                        nr++;
1090                        rth->rt_next = NULL;
1091                        rt_free(rth);
1092                }
1093#if RT_CACHE_DEBUG >= 2
1094                if (nr > 0)
1095                        printk("rt_cache_flush: %d@%02x\n", nr, i);
1096#endif
1097        }
1098#if RT_CACHE_DEBUG >= 1
1099        if (rt_cache_size)
1100        {
1101                printk("rt_cache_flush: bug rt_cache_size=%d\n", rt_cache_size);
1102                rt_cache_size = 0;
1103        }
1104#endif
1105}
1106
1107static void rt_garbage_collect_1(void)
1108{
1109        int i;
1110        unsigned expire = RT_CACHE_TIMEOUT>>1;
```

```
1111        struct rtable * rth, **rthp;
1112        unsigned long now = jiffies;
1113
1114        for (;;)
1115        {
1116                for (i=0; i<RT_HASH_DIVISOR; i++)
1117                {
1118                        if (!ip_rt_hash_table[i])
1119                                continue;
1120                        for (rthp=&ip_rt_hash_table[i]; (rth=*rthp); rthp=&rth->rt_next)
1121                        {
1122                                if (rth->rt_lastuse + expire*(rth->rt_refcnt+1) > now)
1123                                        continue;
1124                                rt_cache_size--;
1125                                cli();
1126                                *rthp=rth->rt_next;
1127                                rth->rt_next = NULL;
1128                                sti();
1129                                rt_free(rth);
1130                                break;
1131                        }
1132                }
1133                if (rt_cache_size < RT_CACHE_SIZE_MAX)
1134                        return;
1135                expire >>= 1;
1136        }
1137}
1138
1139static __inline__ void rt_req_enqueue(struct rt_req **q, struct rt_req *rtr)
1140{
1141        unsigned long flags;
1142        struct rt_req * tail;
1143
1144        save_flags(flags);
1145        cli();
1146        tail = *q;
1147        if (!tail)
1148                rtr->rtr_next = rtr;
1149        else
1150        {
1151                rtr->rtr_next = tail->rtr_next;
1152                tail->rtr_next = rtr;
1153        }
1154        *q = rtr;
1155        restore_flags(flags);
1156        return;
1157}
1158
1159/*
1160 * Caller should mask interrupts.
1161 */
```

```
1162
1163static __inline__ struct rt_req * rt_req_dequeue(struct rt_req **q)
1164{
1165	struct rt_req * rtr;
1166
1167	if (*q)
1168	{
1169		rtr = (*q)->rtr_next;
1170		(*q)->rtr_next = rtr->rtr_next;
1171		if (rtr->rtr_next == rtr)
1172			*q = NULL;
1173		rtr->rtr_next = NULL;
1174		return rtr;
1175	}
1176	return NULL;
1177}
1178
1179/*
1180  Called with masked interrupts
1181 */
1182
1183static void rt_kick_backlog()
1184{
1185	if (!ip_rt_lock)
1186	{
1187		struct rt_req * rtr;
1188
1189		ip_rt_fast_lock();
1190
1191		while ((rtr = rt_req_dequeue(&rt_backlog)) != NULL)
1192		{
1193			sti();
1194			rt_redirect_1(rtr->dst, rtr->gw, rtr->dev);
1195			kfree_s(rtr, sizeof(struct rt_req));
1196			cli();
1197		}
1198
1199		ip_rt_bh_mask &= ~RT_BH_REDIRECT;
1200
1201		ip_rt_fast_unlock();
1202	}
1203}
1204
1205/*
1206 * rt_{del|add|flush} called only from USER process. Waiting is OK.
1207 */
1208
1209static int rt_del(__u32 dst, __u32 mask,
1210		struct device * dev, __u32 gtw, short rt_flags, short metric)
1211{
1212	int retval;
```

```
1213
1214        while (ip_rt_lock)
1215                sleep_on(&rt_wait);
1216        ip_rt_fast_lock();
1217        retval = fib_del_1(dst, mask, dev, gtw, rt_flags, metric);
1218        ip_rt_unlock();
1219        wake_up(&rt_wait);
1220        return retval;
1221}
1222
1223static void rt_add(short flags, __u32 dst, __u32 mask,
1224        __u32 gw, struct device *dev, unsigned short mss,
1225        unsigned long window, unsigned short irtt, short metric)
1226{
1227        while (ip_rt_lock)
1228                sleep_on(&rt_wait);
1229        ip_rt_fast_lock();
1230        fib_add_1(flags, dst, mask, gw, dev, mss, window, irtt, metric);
1231        ip_rt_unlock();
1232        wake_up(&rt_wait);
1233}
1234
1235void ip_rt_flush(struct device *dev)
1236{
1237        while (ip_rt_lock)
1238                sleep_on(&rt_wait);
1239        ip_rt_fast_lock();
1240        fib_flush_1(dev);
1241        ip_rt_unlock();
1242        wake_up(&rt_wait);
1243}
1244
1245/*
1246   Called by ICMP module.
1247 */
1248
1249void ip_rt_redirect(__u32 src, __u32 dst, __u32 gw, struct device *dev)
1250{
1251        struct rt_req * rtr;
1252        struct rtable * rt;
1253
1254        rt = ip_rt_route(dst, 0);
1255        if (!rt)
1256                return;
1257
1258        if (rt->rt_gateway != src ||
1259            rt->rt_dev != dev ||
1260            ((gw^dev->pa_addr)&dev->pa_mask) ||
1261            ip_chk_addr(gw))
1262        {
1263                ip_rt_put(rt);
```

```
1264            return;
1265    }
1266    ip_rt_put(rt);
1267
1268    ip_rt_fast_lock();
1269    if (ip_rt_lock == 1)
1270    {
1271            rt_redirect_1(dst, gw, dev);
1272            ip_rt_unlock();
1273            return;
1274    }
1275
1276    rtr = kmalloc(sizeof(struct rt_req), GFP_ATOMIC);
1277    if (rtr)
1278    {
1279            rtr->dst = dst;
1280            rtr->gw = gw;
1281            rtr->dev = dev;
1282            rt_req_enqueue(&rt_backlog, rtr);
1283            ip_rt_bh_mask |= RT_BH_REDIRECT;
1284    }
1285    ip_rt_unlock();
1286}
1287
1288
1289static __inline__ void rt_garbage_collect(void)
1290{
1291    if (ip_rt_lock == 1)
1292    {
1293            rt_garbage_collect_1();
1294            return;
1295    }
1296    ip_rt_bh_mask |= RT_BH_GARBAGE_COLLECT;
1297}
1298
1299static void rt_cache_add(unsigned hash, struct rtable * rth)
1300{
1301    unsigned long   flags;
1302    struct rtable   **rthp;
1303    __u32           daddr = rth->rt_dst;
1304    unsigned long   now = jiffies;
1305
1306#if RT_CACHE_DEBUG >= 2
1307    if (ip_rt_lock != 1)
1308    {
1309            printk("rt_cache_add: ip_rt_lock==%d\n", ip_rt_lock);
1310            return;
1311    }
1312#endif
1313
1314    save_flags(flags);
```

```
1315
1316        if (rth->rt_dev->header_cache_bind)
1317        {
1318                struct rtable * rtg = rth;
1319
1320                if (rth->rt_gateway != daddr)
1321                {
1322                        ip_rt_fast_unlock();
1323                        rtg = ip_rt_route(rth->rt_gateway, 0);
1324                        ip_rt_fast_lock();
1325                }
1326
1327                if (rtg)
1328                {
1329                        if (rtg == rth)
1330                                rtg->rt_dev->header_cache_bind(&rtg->rt_hh, rtg->rt_dev, ETH_P_IP, rtg-
>rt_dst);
1331                        else
1332                        {
1333                                if (rtg->rt_hh)
1334                                        atomic_inc(&rtg->rt_hh->hh_refcnt);
1335                                rth->rt_hh = rtg->rt_hh;
1336                                ip_rt_put(rtg);
1337                        }
1338                }
1339        }
1340
1341        if (rt_cache_size >= RT_CACHE_SIZE_MAX)
1342                rt_garbage_collect();
1343
1344        cli();
1345        rth->rt_next = ip_rt_hash_table[hash];
1346#if RT_CACHE_DEBUG >= 2
1347        if (rth->rt_next)
1348        {
1349                struct rtable * trth;
1350                printk("rt_cache @%02x: %08x", hash, daddr);
1351                for (trth=rth->rt_next; trth; trth=trth->rt_next)
1352                        printk(" . %08x", trth->rt_dst);
1353                printk("\n");
1354        }
1355#endif
1356        ip_rt_hash_table[hash] = rth;
1357        rthp = &rth->rt_next;
1358        sti();
1359        rt_cache_size++;
1360
1361        /*
1362         * Cleanup duplicate (and aged off) entries.
1363         */
1364
```

```
1365        while ((rth = *rthp) != NULL)
1366        {
1367
1368                cli();
1369                if ((!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT < now)
1370                    || rth->rt_dst == daddr)
1371                {
1372                        *rthp = rth->rt_next;
1373                        rt_cache_size--;
1374                        sti();
1375#if RT_CACHE_DEBUG >= 2
1376                        printk("rt_cache clean %02x@%08x\n", hash, rth->rt_dst);
1377#endif
1378                        rt_free(rth);
1379                        continue;
1380                }
1381                sti();
1382                rthp = &rth->rt_next;
1383        }
1384        restore_flags(flags);
1385}
1386
1387/*
1388   RT should be already locked.
1389
1390   We could improve this by keeping a chain of say 32 struct rtable's
1391   last freed for fast recycling.
1392
1393 */
1394
1395struct rtable * ip_rt_slow_route (__u32 daddr, int local)
1396{
1397        unsigned hash = ip_rt_hash_code(daddr)^local;
1398        struct rtable * rth;
1399        struct fib_node * f;
1400        struct fib_info * fi;
1401        __u32 saddr;
1402
1403#if RT_CACHE_DEBUG >= 2
1404        printk("rt_cache miss @%08x\n", daddr);
1405#endif
1406
1407        rth = kmalloc(sizeof(struct rtable), GFP_ATOMIC);
1408        if (!rth)
1409        {
1410                ip_rt_unlock();
1411                return NULL;
1412        }
1413
1414        if (local)
1415                f = fib_lookup_local(daddr);
```

```
1416        else
1417                f = fib_lookup (daddr);
1418
1419        if (f)
1420        {
1421                fi = f->fib_info;
1422                f->fib_use++;
1423        }
1424
1425        if (!f || (fi->fib_flags & RTF_REJECT))
1426        {
1427#ifdef CONFIG_KERNELD
1428                char wanted_route[20];
1429#endif
1430#if RT_CACHE_DEBUG >= 2
1431                printk("rt_route failed @%08x\n", daddr);
1432#endif
1433                ip_rt_unlock();
1434                kfree_s(rth, sizeof(struct rtable));
1435#ifdef CONFIG_KERNELD
1436                daddr=ntohl(daddr);
1437                sprintf(wanted_route, "%d.%d.%d.%d",
1438                        (int)(daddr >> 24) & 0xff, (int)(daddr >> 16) & 0xff,
1439                        (int)(daddr >> 8) & 0xff, (int)daddr & 0xff);
1440                kerneld_route(wanted_route);    /* Dynamic route request */
1441#endif
1442                return NULL;
1443        }
1444
1445        saddr = fi->fib_dev->pa_addr;
1446
1447        if (daddr == fi->fib_dev->pa_addr)
1448        {
1449                f->fib_use--;
1450                if ((f = fib_loopback) != NULL)
1451                {
1452                        f->fib_use++;
1453                        fi = f->fib_info;
1454                }
1455        }
1456
1457        if (!f)
1458        {
1459                ip_rt_unlock();
1460                kfree_s(rth, sizeof(struct rtable));
1461                return NULL;
1462        }
1463
1464        rth->rt_dst     = daddr;
1465        rth->rt_src     = saddr;
1466        rth->rt_lastuse = jiffies;
```

```
1467        rth->rt_refcnt  = 1;
1468        rth->rt_use     = 1;
1469        rth->rt_next    = NULL;
1470        rth->rt_hh      = NULL;
1471        rth->rt_gateway = fi->fib_gateway;
1472        rth->rt_dev     = fi->fib_dev;
1473        rth->rt_mtu     = fi->fib_mtu;
1474        rth->rt_window  = fi->fib_window;
1475        rth->rt_irtt    = fi->fib_irtt;
1476        rth->rt_tos     = f->fib_tos;
1477        rth->rt_flags   = fi->fib_flags | RTF_HOST;
1478        if (local)
1479                rth->rt_flags   |= RTF_LOCAL;
1480
1481        if (!(rth->rt_flags & RTF_GATEWAY))
1482                rth->rt_gateway = rth->rt_dst;
1483        /*
1484         *      Multicast or limited broadcast is never gatewayed.
1485         */
1486        if (MULTICAST(daddr) || daddr == 0xFFFFFFFF)
1487                rth->rt_gateway = rth->rt_dst;
1488
1489        if (ip_rt_lock == 1)
1490                rt_cache_add(hash, rth);
1491        else
1492        {
1493                rt_free(rth);
1494#if RT_CACHE_DEBUG >= 1
1495                printk(KERN_DEBUG "rt_cache: route to %08x was born dead\n", daddr);
1496#endif
1497        }
1498
1499        ip_rt_unlock();
1500        return rth;
1501}
1502
1503void ip_rt_put(struct rtable * rt)
1504{
1505        if (rt)
1506                atomic_dec(&rt->rt_refcnt);
1507}
1508
1509struct rtable * ip_rt_route(__u32 daddr, int local)
1510{
1511        struct rtable * rth;
1512
1513        ip_rt_fast_lock();
1514
1515        for (rth=ip_rt_hash_table[ip_rt_hash_code(daddr)^local]; rth; rth=rth->rt_next)
1516        {
1517                if (rth->rt_dst == daddr)
```

```
1518            {
1519                    rth->rt_lastuse = jiffies;
1520                    atomic_inc(&rth->rt_use);
1521                    atomic_inc(&rth->rt_refcnt);
1522                    ip_rt_unlock();
1523                    return rth;
1524            }
1525    }
1526    return ip_rt_slow_route (daddr, local);
1527}
1528
1529/*
1530 *      Process a route add request from the user, or from a kernel
1531 *      task.
1532 */
1533
1534int ip_rt_new(struct rtentry *r)
1535{
1536    int err;
1537    char * devname;
1538    struct device * dev = NULL;
1539    unsigned long flags;
1540    __u32 daddr, mask, gw;
1541    short metric;
1542
1543    /*
1544     *      If a device is specified find it.
1545     */
1546
1547    if ((devname = r->rt_dev) != NULL)
1548    {
1549            err = getname(devname, &devname);
1550            if (err)
1551                    return err;
1552            dev = dev_get(devname);
1553            putname(devname);
1554            if (!dev)
1555                    return -ENODEV;
1556    }
1557
1558    /*
1559     *      If the device isn't INET, don't allow it
1560     */
1561
1562    if (r->rt_dst.sa_family != AF_INET)
1563            return -EAFNOSUPPORT;
1564
1565    /*
1566     *      Make local copies of the important bits
1567     *      We decrement the metric by one for BSD compatibility.
1568     */
```

```
1569
1570    flags = r->rt_flags;
1571    daddr = (__u32) ((struct sockaddr_in *) &r->rt_dst)->sin_addr.s_addr;
1572    mask  = (__u32) ((struct sockaddr_in *) &r->rt_genmask)->sin_addr.s_addr;
1573    gw    = (__u32) ((struct sockaddr_in *) &r->rt_gateway)->sin_addr.s_addr;
1574    metric = r->rt_metric > 0 ? r->rt_metric - 1 : 0;
1575
1576    /*
1577     *      BSD emulation: Permits route add someroute gw one-of-my-addresses
1578     *      to indicate which iface. Not as clean as the nice Linux dev technique
1579     *      but people keep using it...  (and gated likes it ;))
1580     */
1581
1582    if (!dev && (flags & RTF_GATEWAY))
1583    {
1584            struct device *dev2;
1585            for (dev2 = dev_base ; dev2 != NULL ; dev2 = dev2->next)
1586            {
1587                    if ((dev2->flags & IFF_UP) && dev2->pa_addr == gw)
1588                    {
1589                            flags &= ~RTF_GATEWAY;
1590                            dev = dev2;
1591                            break;
1592                    }
1593            }
1594    }
1595
1596    if (flags & RTF_HOST)
1597            mask = 0xffffffff;
1598    else if (mask && r->rt_genmask.sa_family != AF_INET)
1599            return -EAFNOSUPPORT;
1600
1601    if (flags & RTF_GATEWAY)
1602    {
1603            if (r->rt_gateway.sa_family != AF_INET)
1604                    return -EAFNOSUPPORT;
1605
1606            /*
1607             *      Don't try to add a gateway we can't reach..
1608             *      Tunnel devices are exempt from this rule.
1609             */
1610
1611            if (!dev)
1612                    dev = get_gw_dev(gw);
1613            else if (dev != get_gw_dev(gw) && dev->type != ARPHRD_TUNNEL)
1614                    return -EINVAL;
1615            if (!dev)
1616                    return -ENETUNREACH;
1617    }
1618    else
1619    {
```

```
1620            gw = 0;
1621            if (!dev)
1622                    dev = ip_dev_bynet(daddr, mask);
1623            if (!dev)
1624                    return -ENETUNREACH;
1625            if (!mask)
1626            {
1627                    if (((daddr ^ dev->pa_addr) & dev->pa_mask) == 0)
1628                            mask = dev->pa_mask;
1629            }
1630    }
1631
1632#ifndef CONFIG_IP_CLASSLESS
1633    if (!mask)
1634            mask = ip_get_mask(daddr);
1635#endif
1636
1637    if (bad_mask(mask, daddr))
1638            return -EINVAL;
1639
1640    /*
1641     *      Add the route
1642     */
1643
1644    rt_add(flags, daddr, mask, gw, dev, r->rt_mss, r->rt_window, r->rt_irtt, metric);
1645    return 0;
1646}
1647
1648
1649/*
1650 *      Remove a route, as requested by the user.
1651 */
1652
1653int ip_rt_kill(struct rtentry *r)
1654{
1655    struct sockaddr_in *trg;
1656    struct sockaddr_in *msk;
1657    struct sockaddr_in *gtw;
1658    char *devname;
1659    int err;
1660    struct device * dev = NULL;
1661
1662    trg = (struct sockaddr_in *) &r->rt_dst;
1663    msk = (struct sockaddr_in *) &r->rt_genmask;
1664    gtw = (struct sockaddr_in *) &r->rt_gateway;
1665    if ((devname = r->rt_dev) != NULL)
1666    {
1667            err = getname(devname, &devname);
1668            if (err)
1669                    return err;
1670            dev = dev_get(devname);
```

```
1671            putname(devname);
1672            if (!dev)
1673                    return -ENODEV;
1674    }
1675    /*
1676     * metric can become negative here if it wasn't filled in
1677     * but that's a fortunate accident; we really use that in rt_del.
1678     */
1679    err=rt_del((__u32)trg->sin_addr.s_addr, (__u32)msk->sin_addr.s_addr, dev,
1680            (__u32)gtw->sin_addr.s_addr, r->rt_flags, r->rt_metric - 1);
1681    return err;
1682}
1683
1684/*
1685 *      Handle IP routing ioctl calls. These are used to manipulate the routing tables
1686 */
1687
1688int ip_rt_ioctl(unsigned int cmd, void *arg)
1689{
1690     int err;
1691     struct rtentry rt;
1692
1693     switch(cmd)
1694     {
1695             case SIOCADDRT:         /* Add a route */
1696             case SIOCDELRT:         /* Delete a route */
1697                     if (!suser())
1698                             return -EPERM;
1699                     err=verify_area(VERIFY_READ, arg, sizeof(struct rtentry));
1700                     if (err)
1701                             return err;
1702                     memcpy_fromfs(&rt, arg, sizeof(struct rtentry));
1703                     return (cmd == SIOCDELRT) ? ip_rt_kill(&rt) : ip_rt_new(&rt);
1704     }
1705
1706     return -EINVAL;
1707}
1708
1709void ip_rt_advice(struct rtable **rp, int advice)
1710{
1711     /* Thanks! */
1712     return;
1713}
1714
1715void ip_rt_update(int event, struct device *dev)
1716{
1717/*
1718 *      This causes too much grief to do now.
1719 */
1720#ifdef COMING_IN_2_1
1721     if (event == NETDEV_UP)
```

```
1722                rt_add(RTF_HOST|RTF_UP, dev->pa_addr, ~0, 0, dev, 0, 0, 0, 0);
1723        else if (event == NETDEV_DOWN)
1724                rt_del(dev->pa_addr, ~0, dev, 0, RTF_HOST|RTF_UP, 0);
1725#endif
1726}
```