对于本文，我们将从用户层使用 Socket CAN 进行数据发送时，数据从用户空间到底层驱动的整个通信流程，用户层使用 Socket CAN 可参考上一篇文章《对 Socket CAN 的理解（2）——【Socket 的原理及使用】》。

当我们在用户层通过 socket 进行 CAN 数据的发送时，需要进行以下操作：

（1）创建一个套接字 socket，采用 AF_CAN 协议；

（2）将创建的套接字返回描述符 sockfd，绑定到本地的地址；

（3）通过 sendto 系统调用函数进行发送；

sendto 的函数声明如下：

int sendto(int sockfd, const void *msg, intlen,unsigned intflags, const struct sockaddr *to, int tolen);

主要参数说明如下：

sockfd:通过 socket 函数生成的套接字描述符；

msg:该指针指向需要发送数据的缓冲区；

len:是发送数据的长度；

to:目标主机的 IP 地址及端口号信息；

sendto 的系统调用会发送一帧数据报到指定的地址，在 CAN 协议调用之前把该地址移到内核空间和检查用户空间数据域是否可读。

在 net/socket.c 源文件中，sendto 函数的系统调用如下代码：

SYSCALL_DEFINE6(sendto, int, fd, void __user *, buff, size_t, len, unsigned, flags, structsockaddr __user *, addr, int, addr_len)

{

```
structsocket *sock;
structsockaddr_storage address;
interr;
structmsghdr msg;
structiovec iov;
intfput_needed;
if(len > INT_MAX)
        len= INT_MAX;
sock= sockfd_lookup_light(fd, &err, &fput_needed);
if(!sock)
        gotoout;
iov.iov_base= buff;
iov.iov_len= len;
msg.msg_name= NULL;
msg.msg_iov= &iov;
msg.msg_iovlen= 1;
msg.msg_control= NULL;
msg.msg_controllen= 0;
msg.msg_namelen= 0;
/*把用户空间的地址移动到内核空间中*/
```

```
        if(addr) {
                err= move_addr_to_kernel(addr, addr_len,(struct sockaddr *)&a
ddress);
                if(err < 0)
                        gotoout_put;
                msg.msg_name= (struct sockaddr *)&address;
                msg.msg_namelen= addr_len;
        }
        if(sock->file->f_flags & O_NONBLOCK)
                flags|= MSG_DONTWAIT;
        msg.msg_flags= flags;
        err= sock_sendmsg(sock, &msg, len);
}
```

在 sendto 的系统调用（sys_sendto）里，会调用到 **sock_sendmsg()**函数，该函数代码如下：

```
int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
{
        structkiocb iocb;
        structsock_iocb siocb;
```

```
        intret;
        init_sync_kiocb(&iocb,NULL);
        iocb.private= &siocb;
        ret= __sock_sendmsg(&iocb, sock, msg, size);
        if(-EIOCBQUEUED == ret)
                ret= wait_on_sync_kiocb(&iocb);
        returnret;
}
```

接下来调用**__sock_sendmsg()**函数。

```
static inline int __sock_sendmsg(struct kiocb *iocb,struct socket *sock, stru
ct msghdr *msg, size_t size)
{
        interr = security_socket_sendmsg(sock, msg, size);
        returnerr ?: __sock_sendmsg_nosec(iocb, sock, msg, size);
}
```

再往下一步就是**__sock_sendmsg_nosec** 函数。在__sock_sendmsg_nosec()函数中会返回一个 sendmsg 函数指针。

```
static inline int __sock_sendmsg_nosec(struct kiocb*iocb, struct socket *so
ck, struct msghdr *msg, size_t size)
{
        structsock_iocb *si = kiocb_to_siocb(iocb);
```

```
    sock_update_classid(sock->sk);
    si->sock= sock;
    si->scm= NULL;
    si->msg= msg;
    si->size= size;
    returnsock->ops->sendmsg(iocb, sock,msg, size);
}
```

在/net/can/raw.c 源文件中，将 raw_sendmsg 函数地址赋给 sendmsg 函数指针，即在函数__sock_sendmsg_nosec()中 return sock->ops->sendmsg(iocb,sock, msg, size)，返回的函数指针将指向 raw_sendmsg()函数。

```
static const struct proto_ops raw_ops = {
    .family        = PF_CAN,
    .release       = raw_release,
    .bind          = raw_bind,
    .connect       = sock_no_connect,
    .socketpair    = sock_no_socketpair,
    .accept        = sock_no_accept,
    .getname       = raw_getname,
    .poll          = datagram_poll,
    .ioctl         = can_ioctl,    /* use can_ioctl() from af_can.c */
```

```
        .listen         = sock_no_listen,
        .shutdown       = sock_no_shutdown,
        .setsockopt     = raw_setsockopt,
        .getsockopt     = raw_getsockopt,
        .sendmsg        = raw_sendmsg,
        .recvmsg        = raw_recvmsg,
        .mmap           = sock_no_mmap,
        .sendpage       = sock_no_sendpage,
};

static int raw_sendmsg(struct kiocb *iocb, struct socket *sock, struct msgh
dr *msg, size_tsize)
{
        structsock *sk = sock->sk;
        structraw_sock *ro = raw_sk(sk);
        structsk_buff *skb;
        structnet_device *dev;
        intifindex;
        interr;
        if(msg->msg_name) {
```

```
                structsockaddr_can *addr =
                        (structsockaddr_can *)msg->msg_name;
                if(msg->msg_namelen < sizeof(*addr))
                        return-EINVAL;
                if(addr->can_family != AF_CAN)
                        return-EINVAL;
                ifindex= addr->can_ifindex;
        }else
                ifindex= ro->ifindex;
        if(size != sizeof(struct can_frame))
                return-EINVAL;
        dev= dev_get_by_index(&init_net, ifindex);
        if(!dev)
                return-ENXIO;
        skb= sock_alloc_send_skb(sk, size, msg->msg_flags & MSG_DONTW
AIT,
                                 &err);
        if(!skb)
                gotoput_dev;
        err= memcpy_fromiovec(skb_put(skb, size), msg->msg_iov, size);
```

```
        if(err < 0)
                gotofree_skb;
        err= sock_tx_timestamp(sk, &skb_shinfo(skb)->tx_flags);
        if(err < 0)
                gotofree_skb;
        /*to be able to check the received tx sock reference in raw_rcv() */
        skb_shinfo(skb)->tx_flags|= SKBTX_DRV_NEEDS_SK_REF;
        skb->dev= dev;
        skb->sk  = sk;
        err= can_send(skb,ro->loopback);
        dev_put(dev);
        if(err)
                gotosend_failed;
        returnsize;
}
```

在 net/can/af_can.c 源文件中，can_send 函数负责 CAN 协议层的数据传输，即传输一帧 CAN 报文（可选本地回环）。参数 skb 指针指向套接字缓冲区和在数据段的 CAN 帧。loop 参数是在本地 CAN 套接字上为监听者提供回环。

```
int can_send(struct sk_buff *skb, int loop)
{
```

```
structsk_buff *newskb = NULL;
structcan_frame *cf = (struct can_frame *)skb->data;
interr;
if(skb->len != sizeof(struct can_frame) || cf->can_dlc > 8) {
        kfree_skb(skb);
        return-EINVAL;
}
if(skb->dev->type != ARPHRD_CAN) {
        kfree_skb(skb);
        return-EPERM;
}
if(!(skb->dev->flags & IFF_UP)) {
        kfree_skb(skb);
        return-ENETDOWN;
}
skb->protocol= htons(ETH_P_CAN);
skb_reset_network_header(skb);
skb_reset_transport_header(skb);
if(loop) {
        /*local loopback of sent CAN frames */
```

```
        /*indication for the CAN driver: do loopback */
        skb->pkt_type= PACKET_LOOPBACK;
        if(!(skb->dev->flags & IFF_ECHO)) {
                /*
                 * If the interface is not capable to doloopback
                 * itself, we do it here.
                 */
                newskb= skb_clone(skb, GFP_ATOMIC);
                if(!newskb) {
                        kfree_skb(skb);
                        return-ENOMEM;
                }
                newskb->sk= skb->sk;
                newskb->ip_summed= CHECKSUM_UNNECESSARY;
                newskb->pkt_type= PACKET_BROADCAST;
        }
}else {
        /*indication for the CAN driver: no loopback required */
        skb->pkt_type= PACKET_HOST;
}
```

```
        /*send to netdevice */
        err= dev_queue_xmit(skb);
        if(err > 0)
                err= net_xmit_errno(err);
        if(err) {
                kfree_skb(newskb);
                returnerr;
        }
        if(newskb)
                netif_rx_ni(newskb);
        /*update statistics */
        can_stats.tx_frames++;
        can_stats.tx_frames_delta++;
        return0;
}
int dev_queue_xmit(struct sk_buff *skb)
{
        structnet_device *dev = skb->dev;
        structnetdev_queue *txq;
        structQdisc *q;
```

```
        intrc = -ENOMEM;


        /*Disable soft irqs for various locks below. Also
         * stops preemption for RCU.
         */
        rcu_read_lock_bh();


        txq= dev_pick_tx(dev, skb);
        q= rcu_dereference_bh(txq->qdisc);


#ifdef CONFIG_NET_CLS_ACT
        skb->tc_verd= SET_TC_AT(skb->tc_verd, AT_EGRESS);
#endif
        trace_net_dev_queue(skb);
        if(q->enqueue) {
                rc= __dev_xmit_skb(skb, q, dev, txq);
                gotoout;
        }
        if(dev->flags & IFF_UP) {
                intcpu = smp_processor_id(); /* ok because BHs are off */
```

```
if(txq->xmit_lock_owner != cpu) {

        if(__this_cpu_read(xmit_recursion) > RECURSION_LIMIT)
                gotorecursion_alert;

        HARD_TX_LOCK(dev,txq, cpu);

        if(!netif_tx_queue_stopped(txq)) {
                __this_cpu_inc(xmit_recursion);
                rc= dev_hard_start_xmit(skb, dev, txq);
                __this_cpu_dec(xmit_recursion);
                if(dev_xmit_complete(rc)) {
                        HARD_TX_UNLOCK(dev,txq);
                        gotoout;
                }
        }
        HARD_TX_UNLOCK(dev,txq);
        if(net_ratelimit())
                printk(KERN_CRIT"Virtual device %s asks to "
```

```
                                "queue packet!\n", dev->name);
                }else {
                        /*Recursion is detected! It is possible,
                         * unfortunately
                         */
recursion_alert:
                        if(net_ratelimit())
                                printk(KERN_CRIT"Dead loop on virtual device "
                                        "%s, fix it urgently!\n",dev->name);
                }
        }

        rc= -ENETDOWN;
        rcu_read_unlock_bh();

        kfree_skb(skb);
        returnrc;
out:
        rcu_read_unlock_bh();
        returnrc;
```

```
}
int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev, structn
etdev_queue *txq)
{
        conststruct net_device_ops *ops = dev->netdev_ops;
        intrc = NETDEV_TX_OK;
        unsignedint skb_len;
        if(likely(!skb->next)) {
                u32features;
                /*
                 * If device doesn't need skb->dst, releaseit right now while
                 * its hot in this cpu cache
                 */
                if(dev->priv_flags & IFF_XMIT_DST_RELEASE)
                        skb_dst_drop(skb);
                if(!list_empty(&ptype_all))
                        dev_queue_xmit_nit(skb,dev);
                skb_orphan_try(skb);
                features= netif_skb_features(skb);
```

```
if(vlan_tx_tag_present(skb) &&
    !(features & NETIF_F_HW_VLAN_TX)) {
        skb = __vlan_put_tag(skb,vlan_tx_tag_get(skb));
        if(unlikely(!skb))
                gotoout;
        skb->vlan_tci= 0;
}
if(netif_needs_gso(skb, features)) {
        if(unlikely(dev_gso_segment(skb, features)))
                gotoout_kfree_skb;
        if(skb->next)
                gotogso;
}else {
        if(skb_needs_linearize(skb, features) &&
           __skb_linearize(skb))
                gotoout_kfree_skb;
        /*If packet is not checksummed and device does not
         * support checksumming for this protocol,complete
         * checksumming here.
         */
```

```
                if(skb->ip_summed == CHECKSUM_PARTIAL) {
                        skb_set_transport_header(skb,
                                skb_checksum_start_offset(skb));
                        if(!(features & NETIF_F_ALL_CSUM) &&
                            skb_checksum_help(skb))
                                gotoout_kfree_skb;
                }
        }
        skb_len= skb->len;
        rc= ops->ndo_start_xmit(skb, dev);
        trace_net_dev_xmit(skb,rc, dev, skb_len);
        if(rc == NETDEV_TX_OK)
                txq_trans_update(txq);
        returnrc;
    }
}
```

以下开始进行到 CAN 的底层驱动代码了，由于 CAN 驱动是编译进内核中，所以在系统启动时会注册 CAN 驱动，注册 CAN 驱动过程中会初始化 d_can_netdev_ops 结构体变量。在这个过程中，d_can_netdev_ops 结构体变量定义了 3 个函数指针，其中(*ndo_start_xmit)函数指针指向 d_can_start_xmit 函数的入口地址。

```
static const struct net_device_ops d_can_netdev_ops = {
        .ndo_open= d_can_open,
        .ndo_stop= d_can_close,
        .ndo_start_xmit =d_can_start_xmit,
};


static netdev_tx_t d_can_start_xmit(struct sk_buff*skb, structnet_device *dev)
{
        u32msg_obj_no;
        structd_can_priv *priv = netdev_priv(dev);
        structcan_frame *frame = (struct can_frame *)skb->data;
        if(can_dropped_invalid_skb(dev, skb))
                returnNETDEV_TX_OK;
        msg_obj_no= get_tx_next_msg_obj(priv);
        /*prepare message object for transmission */
        d_can_write_msg_object(dev,D_CAN_IF_TX_NUM, frame, msg_obj_no);
         can_put_echo_skb(skb,dev, msg_obj_no - D_CAN_MSG_OBJ_TX_FIRST);
        /*
```

```
         * we have to stop the queue in case of a wraparound or
         * if the next TX message object is still inuse
         */
        priv->tx_next++;
        if(d_can_is_next_tx_obj_busy(priv, get_tx_next_msg_obj(priv)) ||
                ((priv->tx_next& D_CAN_NEXT_MSG_OBJ_MASK) == 0))
                netif_stop_queue(dev);
        returnNETDEV_TX_OK;
}
```

在 d_can_start_xmit()函数中，会调用 d_can_write_msg_object()函数准备消息报文进行传输。

```
static void d_can_write_msg_object(struct net_device *dev, intiface, struct can_frame *frame, int objno)
{
        inti;
        unsignedint id;
        u32dataA = 0;
        u32dataB = 0;
        u32flags = 0;
        structd_can_priv *priv = netdev_priv(dev);
```

```
if(!(frame->can_id & CAN_RTR_FLAG))
        flags|= D_CAN_IF_ARB_DIR_XMIT;

if(frame->can_id & CAN_EFF_FLAG) {
        id= frame->can_id & CAN_EFF_MASK;
        flags|= D_CAN_IF_ARB_MSGXTD;
}else
        id= ((frame->can_id & CAN_SFF_MASK) << 18);

flags|= D_CAN_IF_ARB_MSGVAL;
d_can_write(priv,D_CAN_IFARB(iface), IFX_WRITE_IDR(id) | flags);

for(i = 0; i < frame->can_dlc; i++) {
        if(frame->can_dlc <= 4)
                dataA|= (frame->data[i] << (8 * i));
        else{
                if(i < 4)
                        dataA|= (frame->data[i] << (8 * i));
                else
```

```
                dataB|= (frame->data[i] << (8 * (i - 4)));
        }
}

/*DATA write to Message object registers DATAA and DATAB */
if(frame->can_dlc <= 4)
        d_can_write(priv,D_CAN_IFDATA(iface), dataA);
else{
        d_can_write(priv,D_CAN_IFDATB(iface), dataB);
        d_can_write(priv,D_CAN_IFDATA(iface), dataA);
}

/*enable TX interrupt for this message object */
d_can_write(priv,D_CAN_IFMCTL(iface),
                D_CAN_IF_MCTL_TXIE| D_CAN_IF_MCTL_EOB |
                D_CAN_IF_MCTL_TXRQST| D_CAN_IF_MCTL_NEWDAT |
                frame->can_dlc);

/*Put message data into message RAM */
d_can_object_put(dev,iface, objno, D_CAN_IF_CMD_ALL);
```

}

以上即是作者对 Socket CAN 进行数据发送的理解。接下来，我们将分析 Socket CAN 的数据接收！

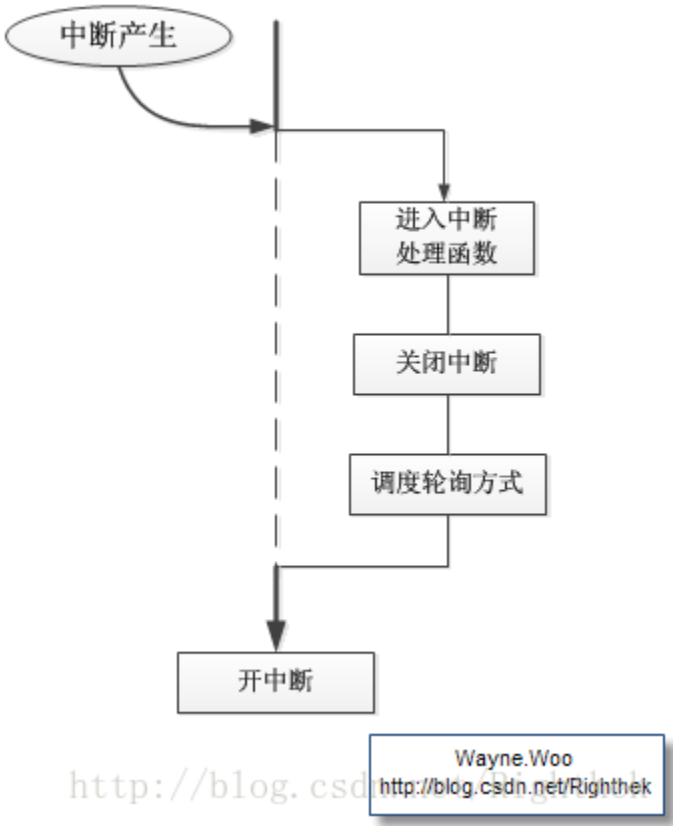
现在我们来分析一下 CAN 总线的接收数据流程，对于网络设备，数据接收大体上采用中断+NAPI 机制进行数据的接收。同样，我们现在的 CAN 模块也是采用同样的方式进行数据的接收。由于我们只针对 CAN 总线接收数据这条主线进行分析。因些，会忽略一些针对 CAN 协议的设置及初始化等相关代码。

在初始化 CAN 设备时，我们需要给 CAN 设备分配 NAPI 功能。我们通过 netif_napi_add()函数将 CAN 设备添加到 NAPI 机制列表中。源码如下：

```
struct net_device *alloc_d_can_dev(intnum_objs)

{

        structnet_device *dev;

        structd_can_priv *priv;

        dev= alloc_candev(sizeof(struct d_can_priv), num_objs/2);

        if(!dev)

                returnNULL;

        priv= netdev_priv(dev);

        netif_napi_add(dev, &priv->napi,d_can_poll, num_objs/2);

        priv->dev= dev;

        priv->can.bittiming_const= &d_can_bittiming_const;

        priv->can.do_set_mode= d_can_set_mode;

        priv->can.do_get_berr_counter= d_can_get_berr_counter;

        priv->can.ctrlmode_supported= (CAN_CTRLMODE_LOOPBACK |
```

```
                                CAN_CTRLMODE_LISTENONLY|

                                CAN_CTRLMODE_BERR_REPORTING|

                                CAN_CTRLMODE_3_SAMPLES);

        returndev;

}
```

以上将 CAN 设备添加到 NAPI 机制列表中后，那么如何去调用它呢？（关于 NAPI 机制，请查看文章《曾经的足迹——对 CAN 驱动中的 NAPI 机制的理解》）接下来就是中断做的事情了。在中断处理函数 d_can_isr 中，我们通过 napi_schedule()函数调度已经在 NAPI 机制列表中的 d_can_poll()函数。该函数会通过轮询的方式接收数据。而根据 NAPI 机制，当中断产生后，会调度轮询机制同时关闭所有的中断。流程如下图：



static irqreturn_t d_can_isr(intirq, void *dev_id)

```
{
        structnet_device *dev = (struct net_device *)dev_id;
        structd_can_priv *priv = netdev_priv(dev);
        priv->irqstatus= d_can_read(priv, D_CAN_INT);
        if(!priv->irqstatus)
                returnIRQ_NONE;
        /*disable all interrupts and schedule the NAPI */
        d_can_interrupts(priv,DISABLE_ALL_INTERRUPTS);
        napi_schedule(&priv->napi);
        returnIRQ_HANDLED;
}
```

当中断产生时，会调用以下函数 d_can_poll()，该函数即采用轮询的方式进行数据的接收。由于 CAN 总线状态中断具有最高优先权，在接收数据之前，需要对 CAN 总线的状态进行判断。而对于 CAN 总线错误状态有三种：

（1） 主动错误；

（2） 被动错误；

（3） 总线关闭；

```
static int d_can_poll(structnapi_struct *napi, int quota)
{
        intlec_type = 0;
        intwork_done = 0;
```

```
structnet_device *dev = napi->dev;
structd_can_priv *priv = netdev_priv(dev);
if(!priv->irqstatus)
        gotoend;
/*status events have the highest priority */
if(priv->irqstatus == STATUS_INTERRUPT) {
        priv->current_status= d_can_read(priv, D_CAN_ES);
        /*handle Tx/Rx events */
        if(priv->current_status & D_CAN_ES_TXOK)
                d_can_write(priv,D_CAN_ES,
                                priv->current_status& ~D_CAN_ES_TXOK);
        if(priv->current_status & D_CAN_ES_RXOK)
                d_can_write(priv,D_CAN_ES,
                                priv->current_status& ~D_CAN_ES_RXOK);
        /*handle state changes */
        if((priv->current_status & D_CAN_ES_EWARN) &&
                        (!(priv->last_status& D_CAN_ES_EWARN))) {
                netdev_dbg(dev,"entered error warning state\n");
                work_done+= d_can_handle_state_change(dev,
                                        D_CAN_ERROR_WARNING);
```

```
}
if((priv->current_status & D_CAN_ES_EPASS) &&
                (!(priv->last_status& D_CAN_ES_EPASS))) {
        netdev_dbg(dev,"entered error passive state\n");
        work_done+= d_can_handle_state_change(dev,
                                D_CAN_ERROR_PASSIVE);
}
if((priv->current_status & D_CAN_ES_BOFF) &&
                (!(priv->last_status& D_CAN_ES_BOFF))) {
        netdev_dbg(dev,"entered bus off state\n");
        work_done +=d_can_handle_state_change(dev,
                                D_CAN_BUS_OFF);
}
/*handle bus recovery events */
if((!(priv->current_status & D_CAN_ES_BOFF)) &&
                (priv->last_status& D_CAN_ES_BOFF)) {
        netdev_dbg(dev,"left bus off state\n");
        priv->can.state= CAN_STATE_ERROR_ACTIVE;
}
if((!(priv->current_status & D_CAN_ES_EPASS)) &&
```

```
                        (priv->last_status& D_CAN_ES_EPASS)) {
                netdev_dbg(dev,"left error passive state\n");
                priv->can.state= CAN_STATE_ERROR_ACTIVE;
            }
            priv->last_status= priv->current_status;
            /*handle lec errors on the bus */
            lec_type= d_can_has_handle_berr(priv);
            if(lec_type)
                work_done+= d_can_handle_bus_err(dev, lec_type);
        }else if ((priv->irqstatus >= D_CAN_MSG_OBJ_RX_FIRST) &&
                (priv->irqstatus<= D_CAN_MSG_OBJ_RX_LAST)) {
            /*handle events corresponding to receive message objects */
            work_done+= d_can_do_rx_poll(dev, (quota - work_done));
        }else if ((priv->irqstatus >= D_CAN_MSG_OBJ_TX_FIRST) &&
                (priv->irqstatus<= D_CAN_MSG_OBJ_TX_LAST)) {
            /*handle events corresponding to transmit message objects */
            d_can_do_tx(dev);
        }
end:
        if(work_done < quota) {
```

```
                napi_complete(napi);
                /*enable all IRQs */
                d_can_interrupts(priv,ENABLE_ALL_INTERRUPTS);
        }
        returnwork_done;
}
```

当总线状态数据状态正常时，进行数据的接收。

```
static int d_can_do_rx_poll(structnet_device *dev, int quota)
{
        structd_can_priv *priv = netdev_priv(dev);
        unsignedint msg_obj, mctrl_reg_val;
        u32num_rx_pkts = 0;
        u32intpnd_x_reg_val;
        u32intpnd_reg_val;
        for(msg_obj = D_CAN_MSG_OBJ_RX_FIRST; msg_obj <= D_CAN_MSG_OBJ_RX_LAST
                        &&quota > 0; msg_obj++) {
                intpnd_x_reg_val= D_CAN_GET_XREG_NUM(priv, D_CAN_INTPND_X);
                intpnd_reg_val= d_can_read(priv,
                                D_CAN_INTPND(intpnd_x_reg_val));
                /*
```

```
 * as interrupt pending register's bit n-1corresponds to
 * message object n, we need to handle the sameproperly.
 */
if(intpnd_reg_val & (1 << (msg_obj - 1))) {
        d_can_object_get(dev,D_CAN_IF_RX_NUM, msg_obj,
                        D_CAN_IF_CMD_ALL&
                        ~D_CAN_IF_CMD_TXRQST);
        mctrl_reg_val= d_can_read(priv,
                        D_CAN_IFMCTL(D_CAN_IF_RX_NUM));
        if(!(mctrl_reg_val & D_CAN_IF_MCTL_NEWDAT))
                continue;
        /*read the data from the message object */
        d_can_read_msg_object(dev, D_CAN_IF_RX_NUM,
                                mctrl_reg_val);
        if(mctrl_reg_val & D_CAN_IF_MCTL_EOB)
                d_can_setup_receive_object(dev,D_CAN_IF_RX_NUM,
                        D_CAN_MSG_OBJ_RX_LAST,0, 0,
                        D_CAN_IF_MCTL_RXIE| D_CAN_IF_MCTL_UMASK
                        |D_CAN_IF_MCTL_EOB);
        if(mctrl_reg_val & D_CAN_IF_MCTL_MSGLST) {
```

```
                d_can_handle_lost_msg_obj(dev,D_CAN_IF_RX_NUM,
                        msg_obj);
                num_rx_pkts++;
                quota--;
                continue;
        }
        if(msg_obj < D_CAN_MSG_OBJ_RX_LOW_LAST)
                d_can_mark_rx_msg_obj(dev,D_CAN_IF_RX_NUM,
                                mctrl_reg_val,msg_obj);
        else if (msg_obj >D_CAN_MSG_OBJ_RX_LOW_LAST)
                /*activate this msg obj */
                d_can_activate_rx_msg_obj(dev,D_CAN_IF_RX_NUM,
                                mctrl_reg_val,msg_obj);
        elseif (msg_obj == D_CAN_MSG_OBJ_RX_LOW_LAST)
                /*activate all lower message objects */
                d_can_activate_all_lower_rx_msg_objs(dev,
                                D_CAN_IF_RX_NUM,mctrl_reg_val);
        num_rx_pkts++;
        quota--;
}
```

```
        }
        returnnum_rx_pkts;
}
```

以下函数是从 CAN 模块的接收寄存器中接收数据。

```
static int d_can_read_msg_object(structnet_device *dev, int iface, int ctrl)
{
        inti;
        u32dataA = 0;
        u32dataB = 0;
        unsignedint arb_val;
        unsignedint mctl_val;
        structd_can_priv *priv = netdev_priv(dev);
        structnet_device_stats *stats = &dev->stats;
        structsk_buff *skb;
        structcan_frame *frame;
        skb= alloc_can_skb(dev, &frame);
        if(!skb) {
                stats->rx_dropped++;
                return-ENOMEM;
        }
```

```
frame->can_dlc= get_can_dlc(ctrl & 0x0F);
arb_val= d_can_read(priv, D_CAN_IFARB(iface));
mctl_val= d_can_read(priv, D_CAN_IFMCTL(iface));
if(arb_val & D_CAN_IF_ARB_MSGXTD)
        frame->can_id= (arb_val & CAN_EFF_MASK) | CAN_EFF_FLAG;
else
        frame->can_id= (arb_val >> 18) & CAN_SFF_MASK;
if(mctl_val & D_CAN_IF_MCTL_RMTEN)
        frame->can_id|= CAN_RTR_FLAG;
else{
        dataA= d_can_read(priv, D_CAN_IFDATA(iface));
        dataB= d_can_read(priv, D_CAN_IFDATB(iface));
        for(i = 0; i < frame->can_dlc; i++) {
                /*Writing MO higher 4 data bytes to skb */
                if(frame->can_dlc <= 4)
                        frame->data[i]= dataA >> (8 * i);
                else{
                        if(i < 4)
                                frame->data[i]= dataA >> (8 * i);
                        else
```

```
                                frame->data[i] = dataB >> (8 *(i-4));
                        }
                }
        }
        netif_receive_skb(skb);
        stats->rx_packets++;
        stats->rx_bytes+= frame->can_dlc;
        return0;
}
```

以上是对底层 CAN 接收数据的分析，并没有涉及到用户空间的调用。