BOSCH M_CAN在Xenomai上的应用

中科时代(深圳)计算机系统有限公司
演讲人：刘杨
日期：2024年11月26日

目录

PART 1

CAN简介

1. 什么是CAN

CAN 是由 Bosch 在 1983 年开发的，用于汽车电子系统中，后来成为工业控制领域的重要通信协议。适用于实时控制和数据传输，支持多节点、多主机通信。

2. CAN的特点

多主结构：CAN网络允许多个节点同时连接，每个节点都可以作为主节点发起通信。
仲裁机制：通过位仲裁机制，确保在同一时间内只有一个节点发送数据，避免冲突。
错误检测：内置多种错误检测和处理机制，确保通信的可靠性。
高速率：支持高达1Mbps的数据传输速率（标准CAN）和5Mbps的数据传输速率（CAN FD）。
灵活的数据帧：支持不同长度的数据帧，标准CAN支持最长8字节的数据，CAN FD支持最长64字节的数据。
低成本：简洁的总线结构，低硬件和布线成本。

3. CAN协议的应用场景

汽车电子：发动机控制单元（ECU）、车身控制、自动驾驶、车载信息娱乐等。
工业自动化：PLC 控制、传感器、执行器、工业机器人等。
医疗设备：医疗监控设备、影像设备等。
楼宇自动化：智能建筑、能源管理、安防系统等。
船舶与航空：船舶控制、飞机系统等。

PART 2

m_can驱动

linux下m_can.c简单分析

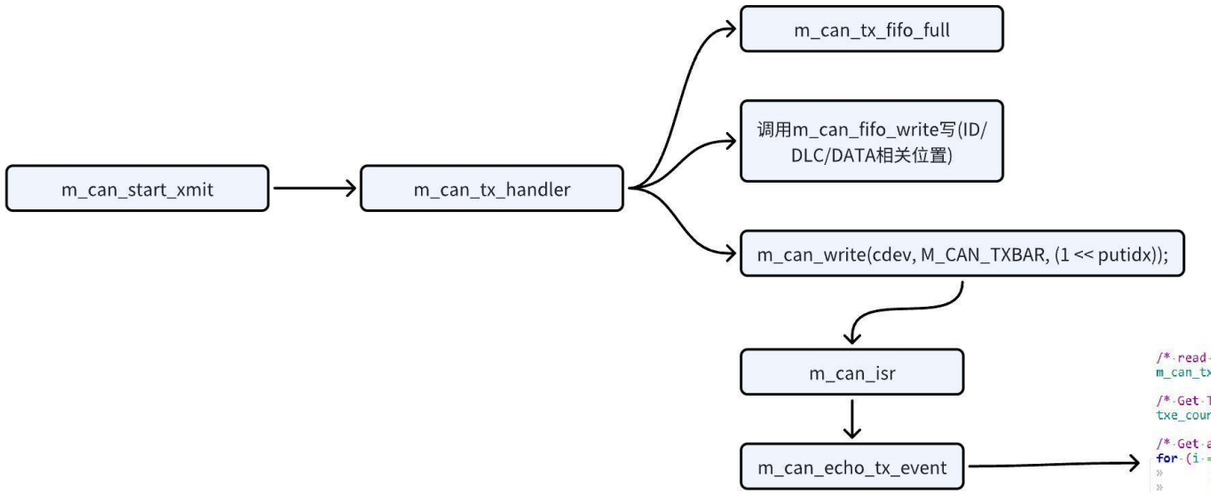
2.1 M_CAN驱动主要涉及的文件

1. drivers/net/can/dev.c //CAN子系统的Core层实现
2. drivers/net/can/m_can/m_can.c //M_CAN通用驱动接口
3. drivers/net/can/m_can/m_can_platform.c //基于M_CAN IP的platform驱动
4. drivers/net/can/dev/rx-offload.c //通过使能NAPI对CAN接收offload
5. net/can/raw.c //SOCK_RAW类型的CAN协议
6. net/can/bcm.c //过滤或发送CAN内容广播管理。
7. net/can/proc.c //procfs文件系统相关

2.2 M_CAN的初始化流程

1. 启动 M_CAN 外设时钟。
2. 配置 M_CAN 控制器的工作模式（正常模式、监听模式、回环模式等）。
3. 配置波特率和时钟源。
4. 配置接收缓冲区（RX FIFO）和发送缓冲区（TX FIFO）。
5. 配置过滤器（如接收过滤器、接受接收的消息等）。
6. 配置中断使能。
7. 其他一些相关配置(如是否生成时间戳等)。

2.3 数据的发送

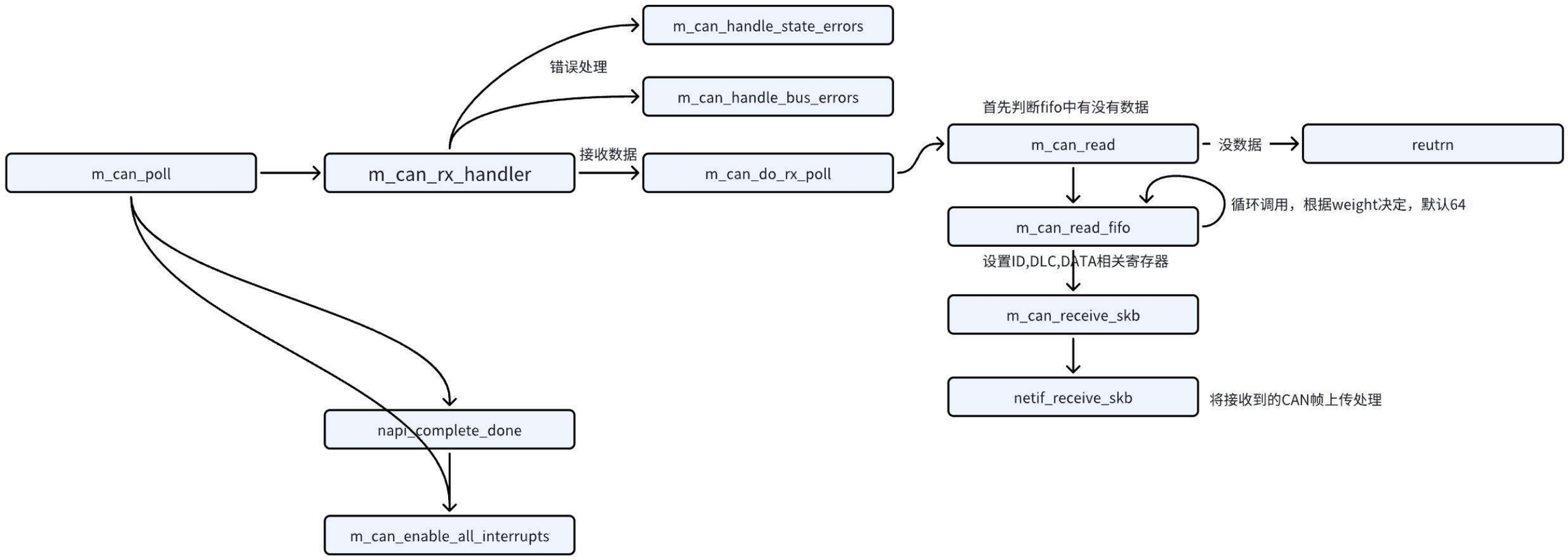



```
/* read tx event fifo status */
m_can_txefs = m_can_read(cdev, M_CAN_TXEFS);

/* Get Tx Event fifo element count */
txe_count = FIELD_GET(TXEFS_EFFL_MASK, m_can_txefs);    //获取有多少个发送事

/* Get and process all sent elements */
for (i = 0; i < txe_count; i++) {
        u32 txe, timestamp = 0;
        int err;

        /* retrieve get index */
        fgi = FIELD_GET(TXEFS_EFGI_MASK, m_can_read(cdev, M_CAN_TXEFS));

        /* get message marker, timestamp */
        err = m_can_txe_fifo_read(cdev, fgi, 4, &txe);
        if (err) {
                netdev_err(dev, "TXE FIFO read returned %d\n", err);
                return err;
        }

        msg_mark = FIELD_GET(TX_EVENT_MM_MASK, txe);
        timestamp = FIELD_GET(TX_EVENT_TXTS_MASK, txe);

        /* ack txe element */
        m_can_write(cdev, M_CAN_TXEFA, FIELD_PREP(TXEFA_EFAI_MASK,
                                                  fgi));

        /* update stats */
        m_can_tx_update_stats(cdev, msg_mark, timestamp);
}  end for i=0;i<txe_count;i++
```

m_can数据发送流程

用户态调用send/write函数之后，最终会调用驱动层的网卡发送函数，对于M_CAN来说，他的发送函数为m_can_start_xmit，m_can_start_xmit函数中先填充ID，然后根据需要填充RTR字段，之后如果FIFO不满的话，依次填充DLC跟DATA位的数据。最后往M_CAN_TXBAR位置写入1，硬件会自动把数据发送出去，之后会产生中断，在中断处理函数中调用m_can_echo_tx_event函数做后续的处理工作，m_can_echo_tx_event函数主要工作是读取M_CAN_TXEFS获取tx event fifo状态，之后获取有多少个发送事件，然后循环的把每个事件发送出去，之后更新下数据发送的状态。

2.4 数据的接收



m_can数据接收流程

数据的发送接收都会有中断产生，对于数据的接收为了提高吞吐量m_can驱动采用了NAPI机制，当网络设备触发中断时，驱动程序会关闭中断，并进入轮询模式，在轮询模式下，驱动程序会批量处理接收到的数据包。处理完成后，驱动程序重新开启中断，等待下一个中断到来。

2.5 部分代码

```
static irqreturn_t m_can_isr(int irq, void *dev_id)
{
        if ((ir & IR_RF0N) || (ir & IR_ERR_ALL_30X)) {
                cdev->irqstatus = ir;
                m_can_disable_all_interrupts(cdev);
                if (!cdev->is_peripheral)
                        napi_schedule(&cdev->napi);
                else if (m_can_rx_peripheral(dev) < 0)
                        goto out_fail;
        }

        if (cdev->version == 30) {
                if (ir & IR_TC) {
                        /* Transmission Complete Interrupt*/
                        u32 timestamp = 0;

                        if (cdev->is_peripheral)
                                timestamp = m_can_get_timestamp(cdev);
                        m_can_tx_update_stats(cdev, 0, timestamp);

                        can_led_event(dev, CAN_LED_EVENT_TX);
                        netif_wake_queue(dev);
                }
        } else {
                if (ir & IR_TEFN) {
                        /* New TX FIFO Element arrived */
                        if (m_can_echo_tx_event(dev) != 0)
                                goto out_fail;

                        can_led_event(dev, CAN_LED_EVENT_TX);
                        if (netif_queue_stopped(dev) &&
                            !m_can_tx_fifo_full(cdev))
                                netif_wake_queue(dev);
                }
        }
```

m_can中断处理流程

```c
static int m_can_tx_handler(struct m_can_classdev *cdev, struct can_frame *cf)
{
	/* Generate ID field for TX buffer Element */
	/* Common to all supported M_CAN versions */
	if (cf->can_id & CAN_EFF_FLAG) {
		fifo_header.id = cf->can_id & CAN_EFF_MASK;
		fifo_header.id |= TX_BUF_XTD;
	} else {
		fifo_header.id = ((cf->can_id & CAN_SFF_MASK) << 18);
	}
	if (cf->can_id & CAN_RTR_FLAG)
		fifo_header.id |= TX_BUF_RTR;
	/* Transmit routine for version >= v3.1.x */
	/* Check if FIFO full */
	if (m_can_tx_fifo_full(cdev)) {
		/* This shouldn't happen */
		rtcandev_warn(dev,
			      "TX queue active although FIFO is full.");
		return 1;
	}
	/* get put index for frame */
	putidx = FIELD_GET(TXFQS_TFQPI_MASK,
			   m_can_read(cdev, M_CAN_TXFQS));
	fifo_header.dlc = FIELD_PREP(TX_BUF_MM_MASK, putidx) |
			  FIELD_PREP(TX_BUF_DLC_MASK, MRAM_CFG_LEN) |
			  TX_BUF_EFC;
	err = m_can_fifo_write(cdev, putidx, M_CAN_FIFO_ID, &fifo_header, 2);
	if (err)
		goto out_fail;
	err = m_can_fifo_write(cdev, putidx, M_CAN_FIFO_DATA,
			       cf->data, DIV_ROUND_UP(cf->can_dlc, 4));
	if (err)
		goto out_fail;
	/* Push loopback echo.
	 * Will be looped back on TX interrupt based on message marker
	 */
	if (rtcan_loopback_pending(dev))
		rtcan_loopback(dev);
	/* Enable TX FIFO element to start transfer  */
	m_can_write(cdev, M_CAN_TXBAR, (1 << putidx));
	/* stop network queue if fifo full */
	if (m_can_tx_fifo_full(cdev)) {
```

```c
static int m_can_echo_tx_event(struct rtcan_device *dev)
{
	u32 txe_count = 0;
	u32 m_can_txefs;
	u32 fgi = 0;
	int i = 0;
	unsigned int msg_mark;

	struct m_can_classdev *cdev = rtcan_priv(dev);

	/* read tx event fifo status */
	m_can_txefs = m_can_read(cdev, M_CAN_TXEFS);

	/* Get Tx Event fifo element count */
	txe_count = FIELD_GET(TXEFS_EFFL_MASK, m_can_txefs);

	/* Get and process all sent elements */
	for (i = 0; i < txe_count; i++) {
		u32 txe, timestamp = 0;
		int err;

		/* retrieve get index */
		fgi = FIELD_GET(TXEFS_EFGI_MASK, m_can_read(cdev, M_CAN_TXEFS));

		/* get message marker, timestamp */
		err = m_can_txe_fifo_read(cdev, fgi, 4, &txe);
		if (err) {
			rtcandev_err(dev, "TXE FIFO read returned %d\n", err);
			return err;
		}

		msg_mark = FIELD_GET(TX_EVENT_MM_MASK, txe);
		timestamp = FIELD_GET(TX_EVENT_TXTS_MASK, txe);

		/* ack txe element */
		m_can_write(cdev, M_CAN_TXEFA, FIELD_PREP(TXEFA_EFAI_MASK,
		            fgi));
	} « end for i=0;i<txe_count;i++ »
```

m_can数据的发送

- 初始化的时候注册了NAPI的回调函数m_can_poll
-

```
if (!cdev->is_peripheral)netif_napi_add(dev,
        &cdev->napi,m_can_poll, M_CAN_NAPI_WEIGHT);
m_can_disable_all_interrupts
napi_schedule-->
        m_can_poll-->
                m_can_rx_handler-->
                        m_can_do_rx_poll-->
        while ((rxfs & RXFS_FFL_MASK) && (quota > 0)) {
                err = m_can_read_fifo(dev, rxfs);
                if (err)
                        return err;
                quota--; //每处理完一包数据减一
                pkts++;  //收到的数据加一
                rxfs = m_can_read(cdev, M_CAN_RXF0S);
        }

        /* Don't re-enable interrupts if the driver had a fatal error
         * (e.g., FIFO read failure).
         */
        if (work_done >= 0 && work_done < quota) {
                napi_complete_done(napi, work_done);
                m_can_enable_all_interrupts(cdev);
        }
```

```
static int m_can_do_rx_poll(struct rtcan_device *dev, int quota)
{
        struct m_can_classdev *cdev = rtcan_priv(dev);
        u32 pkts = 0;
        u32 rxfs;
        int err;

        rxfs = m_can_read(cdev, M_CAN_RXF0S);   //读取Rx FIFO 0 Status 寄存器
        if (!(rxfs & RXFS_FFL_MASK)) {
                rtcandev_dbg(dev, "no messages in fifo0\n");
                return 0;
        }

        // RXFS_FFL_MASK--Number of elements stored in Rx FIFO 0, range 0 to 64.
        while ((rxfs & RXFS_FFL_MASK) && (quota > 0)) {
                err = m_can_read_fifo(dev, rxfs);
                if (err) {
                        return err;
                }

                quota--;
                pkts++;
                rxfs = m_can_read(cdev, M_CAN_RXF0S);
        }

        return pkts;
}«« end m_can_do_rx_poll »»
```

m_can NAPI流程

PART 3

rtcan驱动

m_can.c修改前后对比

3.1 中断部分

```
if ((ir & IR_RF0N) || (ir & IR_ERR_ALL_30X)) {
        cdev->irqstatus = ir;
        m_can_disable_all_interrupts(cdev);
        if (!cdev->is_peripheral)
                napi_schedule(&cdev->napi);
        else if (m_can_rx_peripheral(dev) < 0)
                goto out_fail;
}
```

m_can.c

```
935:     if ((ir & IR_RF0N) || (ir & IR_ERR_ALL_30X)) {
936:             cdev->irqstatus = ir;
937:             {
938:                     ret = m_can_rx_peripheral(dev);
939:                     if (ret < 0)
940:                             goto out_fail;
941:             }
942:     }
```

rtcan_m_can.c

m_can的中断包括接收中断，状态改变中断，bus-err中断，bus-err reporting中断，数据发送完成中断。对于中断中的接收数据以及错误处理部分移除NAPI机制，移除NAPI机制是为了确保系统的低延迟、高确定性。NAPI机制通过poll的方式减少了中断的产生，提高了吞吐量，通过软中断的产生顺序调用注册到全局list中的poll函数，在 NAPI 模式下，数据包需要在 接收队列中等待一段时间，直到 CPU 开始轮询处理。这可能导致数据包的处理延迟增加，影响系统的实时性。移除NAPI机制之后，中断的处理采用同步的处理方式，对于每一包数据的处理更具确定性。

```
} else {
        if (ir & IR_TEFN) {
                /* New TX FIFO Element arrived */
                if (m_can_echo_tx_event(dev) != 0)
                        goto out_fail;

                can_led_event(dev, CAN_LED_EVENT_TX);
                if (netif_queue_stopped(dev) &&
                    !m_can_tx_fifo_full(cdev))
                        netif_wake_queue(dev);
        }
}

if (cdev->is_peripheral)
        can_rx_offload_threaded_irq_finish(&cdev->offl

return IRQ_HANDLED;
```

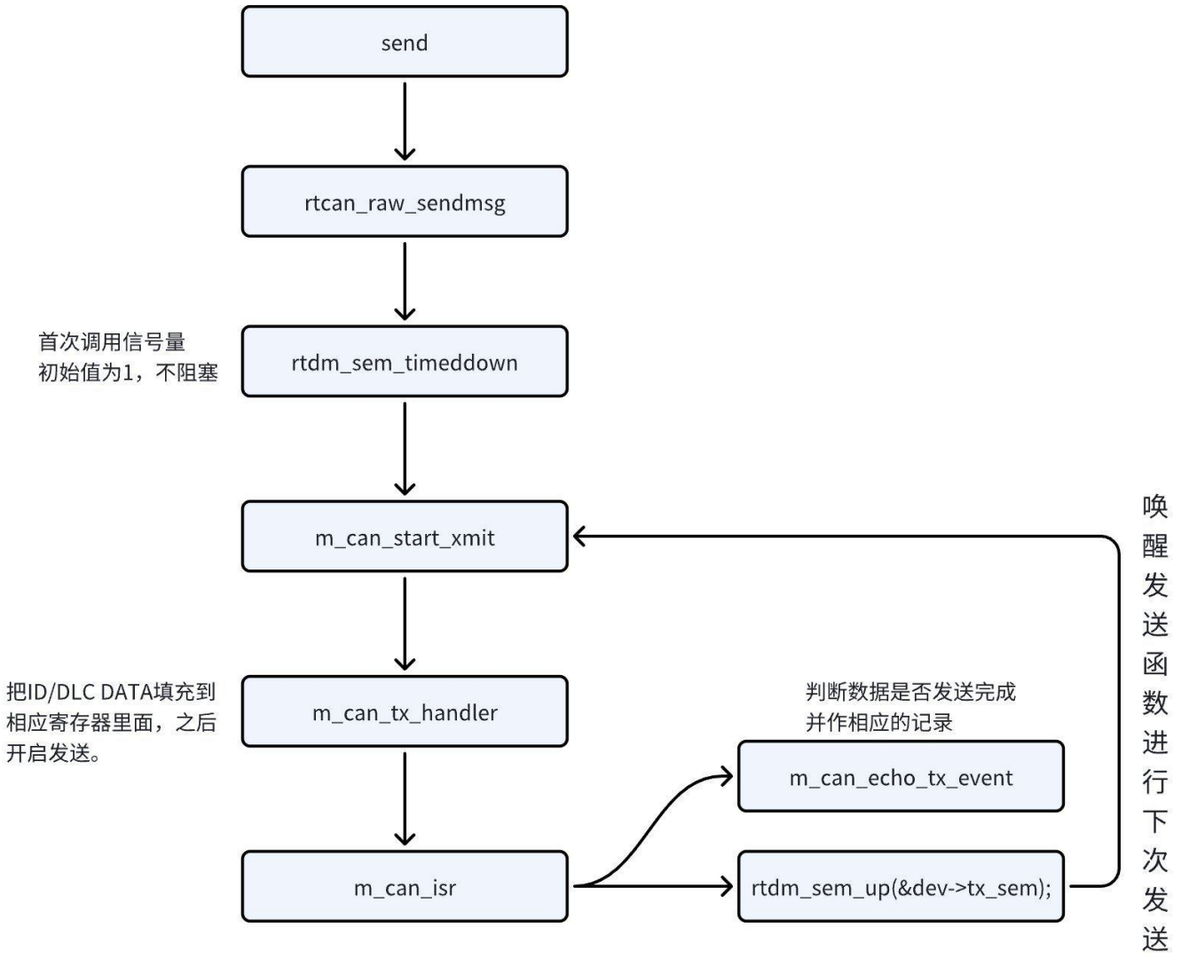
m_can.c

```
946:    if (ir & IR_TEFN) {
947:            /* New TX FIFO Element arrived */
948:            if (m_can_echo_tx_event(dev) != 0)
949:                    goto out_fail;
950:            rtdm_sem_up(&dev->tx_sem);
951:            dev->tx_count++;
952:    }
953:
954:    rtdm_lock_put(&rtcan_socket_lock);
955:    rtdm_lock_put(&rtcan_recv_list_lock);
956:    rtdm_lock_put(&dev->device_lock);
957:    return RTDM_IRQ_HANDLED;
```

rtcan_m_can.c

对于发送中断把唤醒队列改为唤醒发送信号量，RAW_SCOK中的send阻塞等待数据的发送，send被唤醒之后会调用驱动层的hard_start_xmit函数进行数据的发送，数据的发送主要是在hard_start_xmit中完成，中断只是负责读取数据发送状态，之后更新相应的计数。

3.2 数据的发送



rtcan_m_can数据发送流程

首次调用send时信号量为1，之后调用驱动层的hard_start_xmit函数发送数据，此时信号量已经为0，等数据发送完成之后信号量再加1，这样可以保证上一包数据发送完成之后再发送下一包数据。

m_can_tx_hander函数分析：

```
if (m_can_tx_fifo_full(cdev)) {
        /* This shouldn't happen */
        netif_stop_queue(dev);
        netdev_warn(dev,
                    "TX queue active although FIFO is full");

        if (cdev->is_peripheral) {
                kfree_skb(skb);
                dev->stats.tx_dropped++;
                return NETDEV_TX_OK;
        } else {
                return NETDEV_TX_BUSY;
        }
}

/* get put index for frame */
putidx = FIELD_GET(TXFQS_TFQPI_MASK,
                   m_can_read(cdev, M_CAN_TXFQS));

/* Construct DLC Field, with CAN-FD configuration.
 * Use the put index of the fifo as the message marker,
 * used in the TX interrupt for sending the correct echo f
 */

/* get CAN FD configuration of frame */
fdflags = 0;
if (can_is_canfd_skb(skb)) {
        fdflags |= TX_BUF_FDF;
        if (cf->flags & CANFD_BRS)
                fdflags |= TX_BUF_BRS;
}

fifo_header.dlc = FIELD_PREP(TX_BUF_MM_MASK, putidx) |
        FIELD_PREP(TX_BUF_DLC_MASK, can_fd_len2dlc(cf->len
        fdflags | TX_BUF_EFC;
err = m_can_fifo_write(cdev, putidx, M_CAN_FIFO_ID, &fifo_
if (err)
        goto out_fail;

err = m_can_fifo_write(cdev, putidx, M_CAN_FIFO_DATA,
                       cf->data, DIV_ROUND_UP(cf->len, 4));
if (err)
        goto out_fail;

/* Push loopback echo.
 * Will be looped back on TX interrupt based on message ma
 */
can_put_echo_skb(skb, dev, putidx, 0);

/* Enable TX FIFO element to start transfer  */
m_can_write(cdev, M_CAN_TXBAR, (1 << putidx));

/* stop network queue if fifo full */
if (m_can_tx_fifo_full(cdev) ||
```

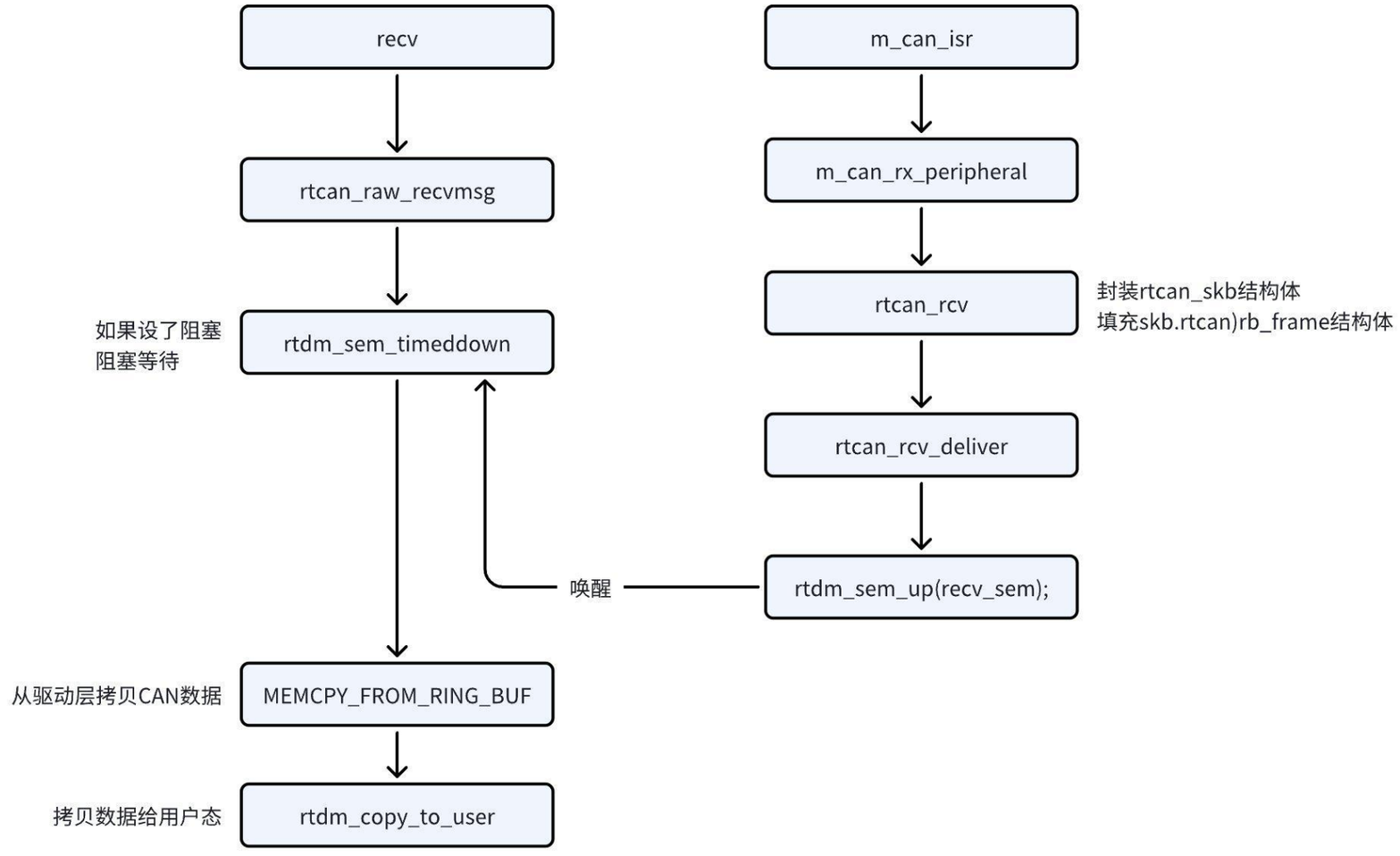
m_can.c

```
1295:       if (m_can_tx_fifo_full(cdev)) {
1296:               /* This shouldn't happen */
1297:               rtcandev_warn(dev,
1298:                             "TX queue active although FIFO is full.");
1299:
1300:               return 1;
1301:       }
1302:
1303:       /* get put index for frame */
1304:       putidx = FIELD_GET(TXFQS_TFQPI_MASK,
1305:                          m_can_read(cdev, M_CAN_TXFQS));
1306:
1307:       /* Construct DLC Field, with CAN-FD configuration.
1308:        * Use the put index of the fifo as the message marker,
1309:        * used in the TX interrupt for sending the correct echo frame.
1310:        */
1311:
1312:       /* get CAN FD configuration of frame */
1313:
1314:       fifo_header.dlc = FIELD_PREP(TX_BUF_MM_MASK, putidx) |
1315:                         FIELD_PREP(TX_BUF_DLC_MASK, MRAM_CFG_LEN) |
1316:                         TX_BUF_EFC;
1317:
1318:       err = m_can_fifo_write(cdev, putidx, M_CAN_FIFO_ID, &fifo_header, 2);
1319:       if (err)
1320:               goto out_fail;
1321:
1322:       err = m_can_fifo_write(cdev, putidx, M_CAN_FIFO_DATA,
1323:                              cf->data, DIV_ROUND_UP(cf->can_dlc, 4));
1324:       if (err)
1325:               goto out_fail;
1326:
1327:       /* Push loopback echo.
1328:        * Will be looped back on TX interrupt based on message marker
1329:        */
1330:       if (rtcan_loopback_pending(dev))
1331:               rtcan_loopback(dev);
1332:
1333:       /* Enable TX FIFO element to start transfer  */
1334:       m_can_write(cdev, M_CAN_TXBAR, (1 << putidx));
1335:
1336:       /* stop network queue if fifo full */
```

rtcan_m_can.c

hard_start_xmit最终会调用m_can_tx_hander，首先获取数据帧的put index位，之后分别填充ID，DLC，DATA位，这样就已经把数据都放到控制器的fifo中了，然后出发一个中断，在中断处理函数中设置tx_sem，最终用户态的send函数就可以进行下一次的数据发送了。

3.3 数据的接收



rtcan数据接收流程

当没有数据到来时如果用户态配置了阻塞接收，会阻塞等待数据的到来，当中断接收到数据后最终rtcan_rcv_deliver函数会把数据拷贝到scok->recv_buf下，之后唤醒用户态程序。

```c
/* napi related */
#define M_CAN_NAPI_WEIGHT		64

static int m_can_do_rx_poll(struct net_device *dev, int quota)
{
	struct m_can_classdev *cdev = netdev_priv(dev);
	u32 pkts = 0;
	u32 rxfs;
	int err;

	rxfs = m_can_read(cdev, M_CAN_RXF0S);
	if (!(rxfs & RXFS_FFL_MASK)) {
		netdev_dbg(dev, "no messages in fifo0\n");
		return 0;
	}

	while ((rxfs & RXFS_FFL_MASK) && (quota > 0)) {
		err = m_can_read_fifo(dev, rxfs);
		if (err)
			return err;


		quota--; //每处理完一包数据减一
		pkts++;  //收到的数据加一
		rxfs = m_can_read(cdev, M_CAN_RXF0S);
	}

	if (pkts)
		can_led_event(dev, CAN_LED_EVENT_RX);

	return pkts;
}
static void m_can_receive_skb(struct m_can_classdev *cdev,
			      struct sk_buff *skb,
			      u32 timestamp)
{
	if (cdev->is_peripheral) {
		struct net_device_stats *stats = &cdev->net->stats;
		int err;

		err = can_rx_offload_queue_sorted(&cdev->offload, skb,
						  timestamp);
		if (err)
			stats->rx_fifo_errors++;
	} else {
		netif_receive_skb(skb);
	}
}
```

m_can.c

```c
/* napi related */
#define M_CAN_NAPI_WEIGHT		1

static int m_can_do_rx_poll(struct rtcan_device *dev, int quota)
{
	struct m_can_classdev *cdev = rtcan_priv(dev);
	u32 pkts = 0;
	u32 rxfs;
	int err;

	rxfs = m_can_read(cdev, M_CAN_RXF0S);
	if (!(rxfs & RXFS_FFL_MASK)) {
		rtcandev_dbg(dev, "no messages in fifo0\n");
		return 0;
	}

	while ((rxfs & RXFS_FFL_MASK) && (quota > 0)) {
		err = m_can_read_fifo(dev, rxfs);
		if (err) {
			return err;
		}

		quota--;
		pkts++;
		rxfs = m_can_read(cdev, M_CAN_RXF0S);
	}



	return pkts;
}
static void m_can_receive_skb(struct m_can_classdev *cdev,
			      struct rtcan_skb *skb,
			      u32 timestamp)
{
	struct rtcan_device *dev = cdev->net;
	rtcan_rcv(dev, skb);



}
```

rtcan_m_can.c

接收数据部分修改为RTDM的RTCAN_RAW中提供的接收函数，最终会调用rtcan_rcv_deliver去处理数据，接收到的数据已经在中断处理时被填充到skb.rb_frame->data中了，rtcan_rcv_deliver从rtcan_skb中拿到数据，之后拷贝到scok->recv_buf下，然后把scok->recv_buf中的数据发送给用户态。

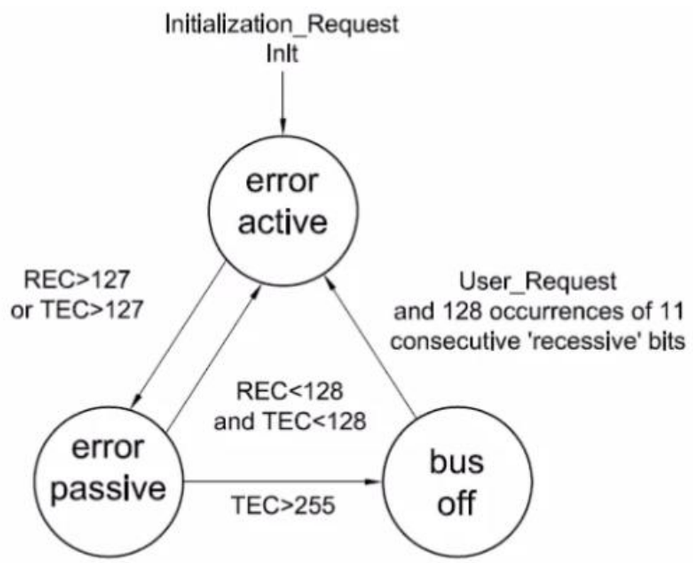
3.4 错误的处理

```
switch (new_state) {
case CAN_STATE_ERROR_WARNING:
        /* error warning state */

        cf->can_id |= CAN_ERR_CRTL;
        cf->data[1] = (bec.txerr > bec.rxerr) ?
                CAN_ERR_CRTL_TX_WARNING :
                CAN_ERR_CRTL_RX_WARNING;
        cf->data[6] = bec.txerr;
        cf->data[7] = bec.rxerr;
        break;
case CAN_STATE_ERROR_PASSIVE:
        /* error passive state */

        cf->can_id |= CAN_ERR_CRTL;
        ecr = m_can_read(cdev, M_CAN_ECR);
        if (ecr & ECR_RP)
                cf->data[1] |= CAN_ERR_CRTL_RX_PASSIVE;
        if (bec.txerr > 127)

                cf->data[1] |= CAN_ERR_CRTL_TX_PASSIVE;
        cf->data[6] = bec.txerr;
        cf->data[7] = bec.rxerr;

        break;
case CAN_STATE_BUS_OFF:
        /* bus-off state */

        cf->can_id |= CAN_ERR_BUSOFF;


        break;
default:
        break;
} end switch new_state

stats->rx_packets++;
stats->rx_bytes += cf->len;

if (cdev->is_peripheral)
        timestamp = m_can_get_timestamp(cdev);

m_can_receive_skb(cdev, skb, timestamp);


return 1;
```

m_can.c

```
662:    switch (new_state) {
663:    case CAN_STATE_ERROR_WARNING:
664:            /* error warning state */
665:            dev->state = CAN_STATE_ERROR_WARNING;
666:            cf->can_id |= CAN_ERR_CRTL;
667:            cf->data[1] = (txerr > rxerr) ?
668:                    CAN_ERR_CRTL_TX_WARNING :
669:                    CAN_ERR_CRTL_RX_WARNING;
670:            cf->data[6] = txerr;
671:            cf->data[7] = rxerr;
672:            break;
673:    case CAN_STATE_ERROR_PASSIVE:
674:            /* error passive state */
675:            dev->state = CAN_STATE_ERROR_PASSIVE;
676:            cf->can_id |= CAN_ERR_CRTL;
677:            if (reg_err_counter & ECR_RP) {
678:                    cf->data[1] |= CAN_ERR_CRTL_RX_PASSIVE;
679:            }
680:            if (txerr > 127) {
681:                    cf->data[1] |= CAN_ERR_CRTL_TX_PASSIVE;
682:            }
683:            cf->data[6] = txerr;
684:            cf->data[7] = rxerr;
685:            break;
686:    case CAN_STATE_BUS_OFF:
687:            /* bus-off state */
688:            dev->state = CAN_STATE_BUS_OFF;
689:            cf->can_id |= CAN_ERR_BUSOFF;
690:            /* destroy waiting senders */
691:            rtdm_sem_destroy(&dev->tx_sem);
692:            break;
693:    default:
694:            break;
695:    } end switch new_state


696:
697:    /* Store the interface index */
698:    cf->can_ifindex = dev->ifindex;
699:    rtcan_rcv(dev, &skb);
700:
701:    return 1;
```

rtcan_m_can.c



当节点的 TEC 或 REC 的值超过 127 时，节点从主动错误状态迁移到被动错误状态。
当节点的 TEC 和 REC 的值都小于 128 时，节点从被动错误状态迁移到主动错误状态。
当节点的 TEC 的值达到 255 时，节点从被动错误状态迁移到 Bus-Off 状态。
当节点从 Bus-Off 状态恢复时，它会重新进入被动错误状态。(节点在 Bus-Off 状态下等待一段时间（通常为 128 个仲裁时间段）后，重新初始化错误计数器，并进入被动错误状态。)

对于主动错误，被动错误以及bus-off的处理，主要把非实时的skb相关的接收函数改为rtcan_rcv(即做数据接收的处理也做错误的计数处理)，把RAW_SCOK层的计数替换成RTDM的。

```c
static int m_can_handle_lost_msg(struct net_device *
{
	struct m_can_classdev *cdev = netdev_priv(dev);
	struct net_device_stats *stats = &dev->stats;
	struct sk_buff *skb;
	struct can_frame *frame;
	u32 timestamp = 0;

	netdev_err(dev, "msg lost in rxf0\n");

	stats->rx_errors++;
	stats->rx_over_errors++;

	skb = alloc_can_err_skb(dev, &frame);
	if (unlikely(!skb))
		return 0;

	frame->can_id |= CAN_ERR_CRTL;
	frame->data[1] = CAN_ERR_CRTL_RX_OVERFLOW;

	if (cdev->is_peripheral)
		timestamp = m_can_get_timestamp(cdev);

	m_can_receive_skb(cdev, skb, timestamp);

	return 1;
}
```

```c
static int m_can_handle_lec_err(struct net_device *d
		enum m_can_lec_type lec_type)
{
	struct m_can_classdev *cdev = netdev_priv(dev);
	struct net_device_stats *stats = &dev->stats;
	struct can_frame *cf;
	struct sk_buff *skb;
	u32 timestamp = 0;

	cdev->can.can_stats.bus_error++;
	stats->rx_errors++;

	/* propagate the error condition to the CAN stack */
	skb = alloc_can_err_skb(dev, &cf);
	if (unlikely(!skb))
		return 0;
```

m_can.c

```c
static int m_can_handle_lost_msg(struct rtcan_device *dev)
{
	struct m_can_classdev *cdev = rtcan_priv(dev);
	struct rtcan_skb skb;
	struct rtcan_rb_frame *frame = &skb.rb_frame;

	u32 timestamp = 0;


	rtcandev_err(dev, "msg lost in rxf0\n");
	skb.rb_frame_size = EMPTY_RB_FRAME_SIZE + CAN_ERR_DLC;



	frame->can_id |= CAN_ERR_CRTL;
	frame->data[1] = CAN_ERR_CRTL_RX_OVERFLOW;
	frame->can_dlc = CAN_ERR_DLC;
	m_can_receive_skb(cdev, &skb, timestamp);


	return 1;
}
```

```c
static int m_can_handle_lec_err(struct rtcan_device *dev,
		enum m_can_lec_type lec_type)
{
	struct m_can_classdev *cdev = rtcan_priv(dev);
	struct rtcan_skb skb;
	struct rtcan_rb_frame *cf = &skb.rb_frame;


	if (lec_type == LEC_UNUSED || lec_type == LEC_NO_ERROR)
		return 0;

	if (cdev->bus_err_on < 2) //rtcan_raw_recvmsg 中会调用do_e


		return 0;

	skb.rb_frame_size = EMPTY_RB_FRAME_SIZE + CAN_ERR_DLC;
	cdev->bus_err_on--;
```

rtcan_m_can.c

对于bus-err错误的处理，读取PSR寄存器获取错误的原因并且记录下来，主要错误包括填充错误，格式错误，ack错误，位错误，CRC错误，把net_devices_stats中的计数统一改为rtcan_device下的err_count。

3.5 用户态设置

① 设置波特率并启动can

rtcanconfig rtcan0 --baudrate=500000 start/up

```c
static const struct can_bittiming_const m_can_data_bittiming_c
        .name = KBUILD_MODNAME,
        .tseg1_min = 1,         /* Time segment 1 = prop_seg +
        .tseg1_max = 32,
        .tseg2_min = 1,         /* Time segment 2 = phase_seg2
        .tseg2_max = 16,
        .sjw_max = 16,
        .brp_min = 1,
        .brp_max = 32,
        .brp_inc = 1,
};

static int m_can_set_bittiming(struct net_device *dev
{
        struct m_can_classdev *cdev = netdev_priv(dev);
        const struct can_bittiming *bt = &cdev->can.bittiming;
        const struct can_bittiming *dbt = &cdev->can.data_bitt
        u16 brp, sjw, tseg1, tseg2;
        u32 reg_btp;

        brp = bt->brp - 1;
        sjw = bt->sjw - 1;
        tseg1 = bt->prop_seg + bt->phase_seg1 - 1;
        tseg2 = bt->phase_seg2 - 1;
        reg_btp = FIELD_PREP(NBTP_NBRP_MASK, brp) |
                  FIELD_PREP(NBTP_NSJW_MASK, sjw) |
                  FIELD_PREP(NBTP_NTSEG1_MASK, tseg1) |
                  FIELD_PREP(NBTP_NTSEG2_MASK, tseg2);
        m_can_write(cdev, M_CAN_NBTP, reg_btp);
```

m_can.c

```c
static int m_can_save_bit_time(struct rtcan_device *dev, struct can_bittime *bt,
                               rtdm_lockctx_t *lock_ctx)
{
        struct m_can_classdev *priv = rtcan_priv(dev);

        memcpy(&priv->bit_time, bt, sizeof(*bt));
        return 0;
}

static int m_can_set_bittiming(struct rtcan_device *dev)
{
        struct m_can_classdev *cdev = rtcan_priv(dev);
        const struct can_bittime *bt = &cdev->bit_time;

        u16 brp, sjw, tseg1, tseg2;
        u32 reg_btp;

        brp = bt->std.brp - 1;
        sjw = bt->std.sjw - 1;
        tseg1 = bt->std.prop_seg + bt->std.phase_seg1 - 1;
        tseg2 = bt->std.phase_seg2 - 1;
        reg_btp = FIELD_PREP(NBTP_NBRP_MASK, brp) |
                  FIELD_PREP(NBTP_NSJW_MASK, sjw) |
                  FIELD_PREP(NBTP_NTSEG1_MASK, tseg1) |
                  FIELD_PREP(NBTP_NTSEG2_MASK, tseg2);
        m_can_write(cdev, M_CAN_NBTP, reg_btp);
```

rtcm_can.c

用户态调用设置波特率时驱动中的do_set_bit_time会被调用，m_can_chip_config被调用时会设置位时序相关参数。

```c
static void m_can_start(struct rtcan_device *dev, rtdm_lockctx_t *lock_ctx)
{
	int err;
	struct m_can_classdev *cdev = rtcan_priv(dev);

	switch (dev->state) {

	case CAN_STATE_ACTIVE:
	case CAN_STATE_BUS_WARNING:
	case CAN_STATE_BUS_PASSIVE:
		rtcandev_info(dev, "Mode start: state active, bus warning, or passive\n");
		break;

	case CAN_STATE_STOPPED:
		err = rtdm_irq_request(&dev->irq_handle, cdev->irq,
				       m_can_isr, RTDM_IRQTYPE_SHARED, "M_CAN",
				       (void *)dev);
		if (err) {
			rtcandev_err(dev, "couldn't request irq %d\n",
				     cdev->irq);
			return;
		}

		/* start chip and queuing */
		m_can_chip_config(dev);

		dev->state = CAN_STATE_ERROR_ACTIVE;

		/* enable status change, error and module interrupts */
		m_can_enable_all_interrupts(cdev);

		/* Set up sender "mutex" */
		rtdm_sem_init(&dev->tx_sem, 1);

		break;
```

用户态调用启动can时(start/up)，do_set_mode会被调用，首次启动时会注册rtdm中断，并且做芯片相关的初始化工作，同时使能中断，初始化信号量。

② 数据的收发

接收端：

- socket(PF_CAN, SOCK_RAW, CAN_RAW); //创建socket
- namecpy(ifr.ifr_name, “rtcan0”); //告诉驱动can接口的名字
- ret = ioctl(s, SIOCGIFINDEX, &ifr); //获取驱动的接口index
- setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER, &err_mask, sizeof(err_mask)); //如果需要过滤设置过滤规则
- recv_addr.can_family = AF_CAN;
  recv_addr.can_ifindex = ifr.ifr_ifindex; //设置CAN 地址族
- bind(s, (struct sockaddr *)&recv_addr, sizeof(struct sockaddr_can)); //绑定
- recvmsg(s, &msg, 0); //接收数据

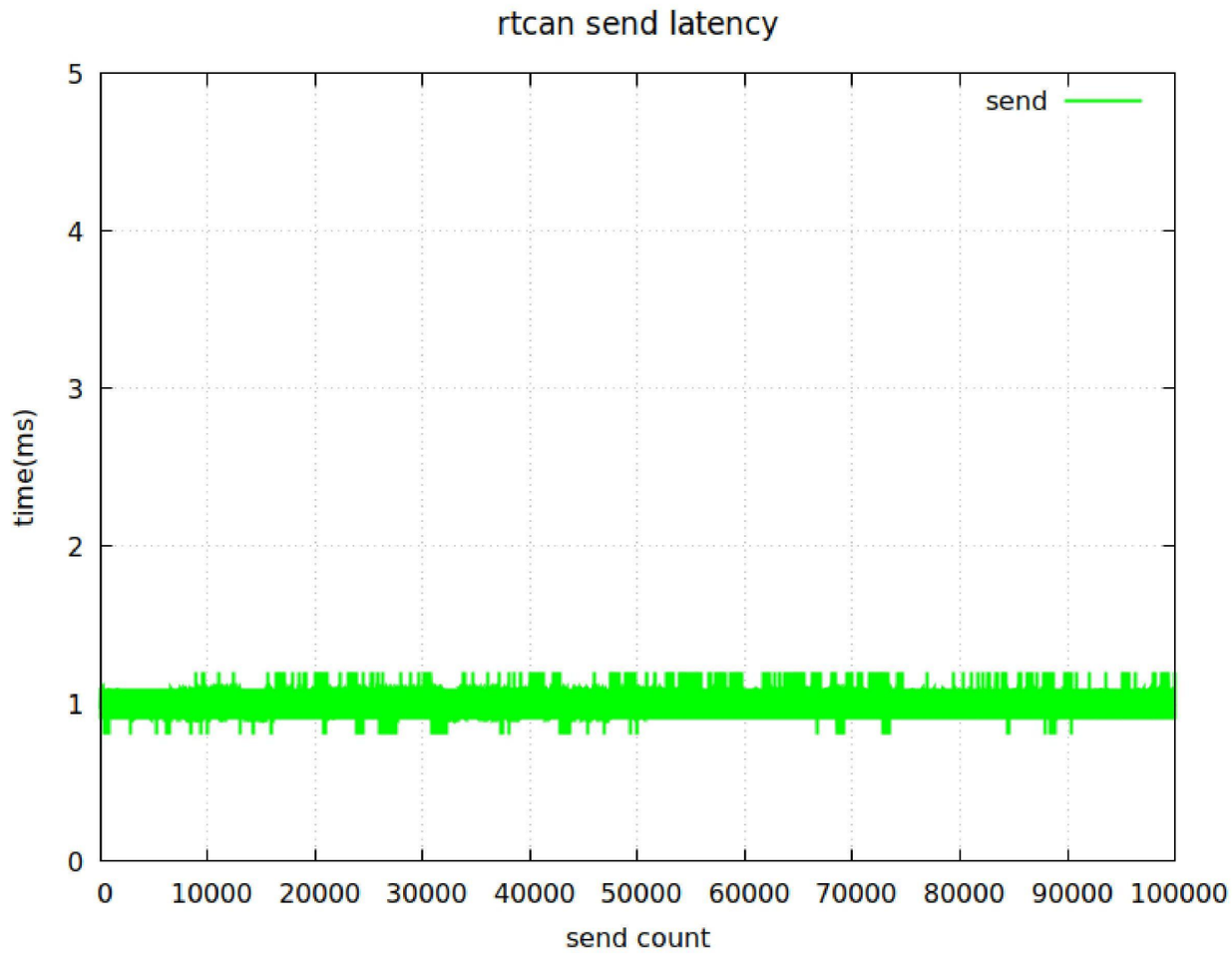
发送端：

- socket(PF_CAN, SOCK_RAW, CAN_RAW); //创建socket
- ioctl(s, SIOCGIFINDEX, &ifr); //获取驱动接口index
- namecpy(ifr.ifr_name, “rtcan0”); //告诉驱动接口名字
- to_addr.can_family = AF_CAN; //设置CAN 地址族
  to_addr.can_ifindex = ifr.ifr_ifindex;
- setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0); //设置过滤规则
- bind(s, (struct sockaddr *)&to_addr, sizeof(to_addr)); //绑定
- send(s, (void *)&frame, sizeof(can_frame_t), 0); //发送数据

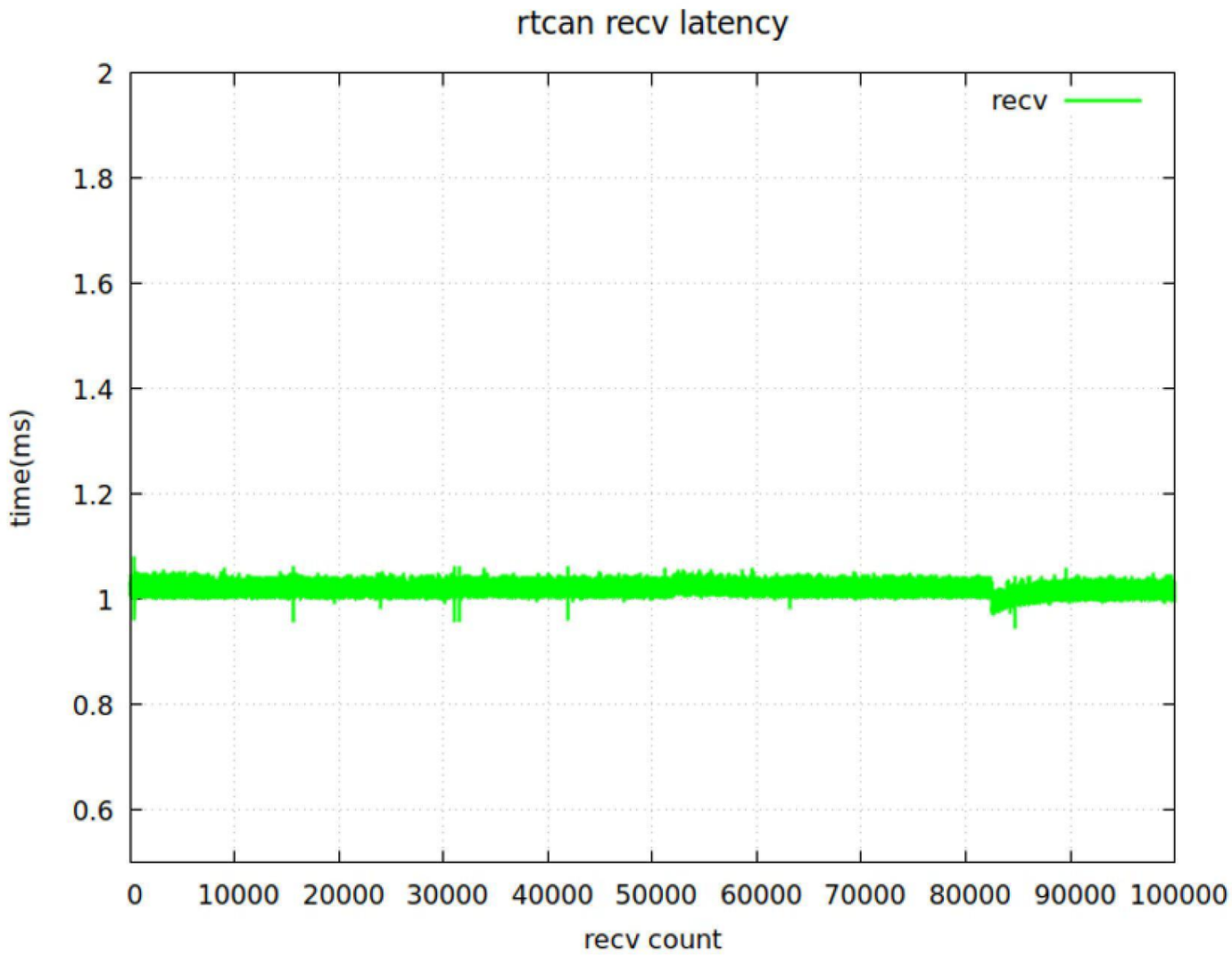
PART 4

测试数据

1. 部分测试数据



发送端采用实时rtcan驱动每1ms发送8字节数据，接收端采用USB转CAN模块接收数据，每次接收到数据之后打印时间戳，统计前后两次时间戳的差值。



发送端跟接收端都采用rtcan驱动程序，发送端没1ms发送8字节数据，在接收端接收到数据的时候打印时间戳，统计前后两次时间戳的差值。

SINSEGYE

感谢聆听