The Embedded I/O Company

TEWS TECHNOLOGIES

TIP903-SW-82

Linux Device Driver

3 Channel Extended CAN Bus IP

Version 1.2.x

User Manual

Issue 1.2.3

November 2010

TEWS TECHNOLOGIES GmbH

Am Bahnhof 7 25469 Halstenbek, Germany

Phone: +49 (0) 4101 4058 0 Fax: +49 (0) 4101 4058 19

e-mail: info@tews.com www.tews.com

TIP903-SW-82

Linux Device Driver

3 Channel Extended CAN Bus IP

Issue	Description	Date
1.0	First Issue	October 2, 2002
1.1	Support for DEVFS and SMP	February 25, 2004
1.2.0	Introduction and installation description modified	April 05, 2006
1.2.1	UDEV description added	December 11, 2007
1.2.2	Carrier Driver description added	July 7, 2008
1.2.3	General revision, Address TEWS LLC removed Avoiding undefined symbols warning note	November 25, 2010

Table of Contents

1 Introduction

1.1 Device Driver

The TIP903-SW-82 Linux device driver allows the operation of TIP903 IPAC modules on Linux operating systems.

Because the TIP903 device driver is stacked on the TEWS TECHNOLOGIES IPAC carrier driver, it's necessary to install also the appropriate IPAC carrier driver. Please refer to the IPAC carrier driver user manual for further information.

The TIP903 device driver includes the following features:

- Transmission and receive of Standard and extended Identifiers
- Up to 15 receive message queues with user defined size
- Variable allocation of receive message objects to receive queues
- Separate task queues for each receive queue and transmission buffer message object
- Standard bit rates from 20 kbit up to 1.0 Mbit and user defined bit rates
- Message acceptance filtering
- Definition of receive and remote buffer message objects
- Transmission and receive of Standard and extended Identifiers
- Designed as Linux kernel module with dynamically loading (modprobe*)*.
- Supports shared IRQ's.
- Creates devices with dynamically allocated or fixed major device numbers.

The TIP903-SW-82 supports the modules listed below:

TIP903-10	3 channel extended CAN bus IndustryPack	(IndustryPack®)

To get more information about the features and use of the supported devices it is recommended to read the manuals listed below.

TIP903 User manual

TIP903 Engineering Manual

Architectural Overview of the Intel 82527 CAN controller (part of TIP903 Engineering Manual)

CARRIER-SW-82 IPAC Carrier User Manual

1.2 IPAC Carrier Driver

IndustryPack (IPAC) carrier boards have different implementations of the system to IndustryPack bus bridge logic, different implementations of interrupt and error handling and so on. Also the different byte ordering (big-endian versus little-endian) of CPU boards will cause problems on accessing the IndustryPack I/O and memory spaces.

To simplify the implementation of IPAC device drivers which work with any supported carrier board, TEWS TECHNOLOGIES has designed a so called Carrier Driver that hides all differences of different carrier boards under a well defined interface.

The TEWS TECHNOLOGIES IPAC Carrier Driver CARRIER-SW-82 is part of this TIP903-SW-82 distribution. It is located in directory CARRIER-SW-82 on the corresponding distribution media.

This IPAC Device Driver requires a properly installed IPAC Carrier Driver. Due to the design of the Carrier Driver, it is sufficient to install the IPAC Carrier Driver once, even if multiple IPAC Device Drivers are used.

Please refer to the CARRIER-SW-82 User Manual for a detailed description how to install and setup the CARRIER-SW-82 device driver, and for a description of the TEWS TECHNOLOGIES IPAC Carrier Driver concept.

2 Installation

The directory TIP903-SW-82 on the distribution media contains the following files:

TIP903-SW-82-1.2.3.pdf	This manual in PDF format
TIP903-SW-82-SRC.tar.gz	GZIP compressed archive with driver source code
ChangeLog.txt	Release history
Release.txt	Release information

The GZIP compressed archive TIP903-SW-82.tar.gz contains the following files and directories:

tip903/tip903.c	Driver source code
tip903/tip903def.h	Driver include file
tip903/tip903.h	Driver include file for application program
tip903/I82527.h	Intel 82527 CAN controller definitions
tip903/makenode	Script to create device nodes on the file system
tip903/Makefile	Device driver make file
tip903/example/tip903exa.c	Example application
tip903/example/Makefile	Example application make file
tip903/include/config.h	Kernel independent header file
tip903/include/tpmodule.h	Kernel independent library header file
tip903/include/tpmodule.c	Kernel independent library source code file

In order to perform an installation, extract all files of the archive TIP903-SW-82-SRC.tar.gz to the desired target directory. The command 'tar -xzvf TIP903-SW-82-SRC.tar.gz' will extract the files into the local directory.

Before building a new device driver, the TEWS TECHNOLOGIES IPAC carrier driver must be installed properly, because this driver includes the header file *ipac_carrier.h*, which is part of the IPAC carrier driver distribution. Please refer to the IPAC carrier driver user manual in the directory path *CARRIER-SW-82* on the separate distribution media.

2.1 Build and install the device driver

- Login as *root*
- Change to the target directory
- Copy file tip903.h to /usr/include/ to allow user application access
- To create and install the driver in the module directory */lib/modules/<version>/misc* enter:

  # make install

  For Linux kernel 2.6.x, there may be compiler warnings claiming some undefined ipac_* symbols. These warnings are caused by exported symbols of the IPAC carrier driver, which are unknown during compilation of this TIP driver.

  These warnings can be ignored or avoided by defining the path where the symbol definitions can be found (works only for kernel versions >2.6.28).

  To avoid these warnings the symbol KBUILD_EXTRA_SYMBOLS can be added to the make command line and must specify the absolute path where the symbols of the IPAC carrier driver can be found. For example:
  make install KBUILD_EXTRA_SYMBOLS=/usr/ipac_carrier/class/Module.symvers

- Also after the first build we have to execute *depmod* to create a new dependency description for loadable kernel modules. This dependency file is later used by *modprobe* to automatically load the correct IPAC carrier driver modules.

  # depmod –aq

2.2 Uninstall the device driver

- Login as *root*
- Change to the target directory
- To remove the driver from the module directory */lib/modules/<version>/misc* enter:

  # make uninstall

- Update kernel module dependency description file

  # depmod –aq

2.3 Install device driver into the running kernel

- To load the device driver into the running kernel, login as root and execute the following commands:

  # modprobe tip903drv

- After the first build or if you are using dynamic major device allocation it's necessary to create new device nodes on the file system. Please execute the script file *makenode* to do this. If your kernel has enabled a device file system (devfs or sysfs with udev) then you have to skip running the *makenode* script. Instead of creating device nodes from the script the driver itself takes creating and destroying of device nodes in its responsibility.

  # sh makenode

On success the device driver will create a minor device for each TIP903 CAN channel found. The first TIP903 channel can be accessed using device node /dev/tip903_0, the second TIP903 channel with device node /dev/tip903_1, the third channel TIP903 with device node /dev/tip903_2 and so on.

The allocation of device nodes to physical TIP903 modules depends on the search order of the IPAC carrier driver. Please refer to the IPAC carrier user manual.

Loading of the TIP903 device driver will only work if kernel KMOD support is installed, necessary carrier board drivers already installed and the kernel dependency file is up to date. If KMOD support isn't available you have to build either a new kernel with KMOD installed or you have to install the IPAC carrier kernel modules manually in the correct order (please refer to the IPAC carrier driver user manual).

2.4 Remove device driver from the running kernel

- To remove the device driver from the running kernel login as root and execute the following command:

  # **modprobe tip903drv –r**

If your kernel has enabled devfs or sysfs (udev), all /dev/tip903_x nodes will be automatically removed from your file system after this.

Be sure that the driver isn't opened by any application program. If opened you will get the response "*tip903drv: Device or resource busy*" and the driver will still remain in the system until you close all opened files and execute *modprobe –r* again.

2.5 Change Major Device Number

The TIP903 driver use dynamic allocation of major device numbers by default. If this isn't suitable for the application it's possible to define a major number for the driver. If the kernel has enabled devfs the driver will not use the symbol TIP903_MAJOR.

To change the major number edit the file tip903drv.c, change the following symbol to appropriate value and enter **make install** to create a new driver.

TIP903_MAJOR	Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TIP903_MAJOR       122
```

2.6 Receive Queue Configuration

Received CAN messages will be stored in receive queues. Each receive queue contains a FIFO and a separate task wait queue. The number of receive queues and the depth of the FIFO can be adapted by changing the following symbols in *tip903.c*.

NUM_RX_QUEUES	Defines the number of receive queues for each device (default = 3). Valid numbers are in range between 1 and 15. To support multitasking and multiprocessing the driver allows only one thread per queue. So NUM_RX_QUEUES is the same as the maximum amount of concurrent threads during reading.
RX_FIFO_SIZE	Defines the depth of the message FIFO inside each receive queue (default = 100). Valid numbers are in range between 1 and MAXINT.

3 Device Input/Output functions

This chapter describes the interface to the device driver I/O system.

3.1 open()

NAME

open() - open a file descriptor

SYNOPSIS

#include <fcntl.h>

int open (const char *filename, int flags)

DESCRIPTION

The open function creates and returns a new file descriptor for the file named by *filename*. The *flags* argument controls how the file is to be opened. This is a bit mask; you create the value by the bitwise OR of the appropriate parameters (using the | operator in C). See also the GNU C Library documentation for more information about the open function and open flags.

EXAMPLE

```
int fd;

fd = open(“/dev/tip903_0”, O_RDWR);
if (fd == -1)
{
  /* handle error condition */
}
```

RETURNS

The normal return value from open is a non-negative integer file descriptor. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

ENODEV	The requested minor device does not exist.

This is the only error code returned by the driver, other codes may be returned by the I/O system during open.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.2 close()

NAME

close() – close a file descriptor

SYNOPSIS

#include <unistd.h>

int **close** (int *filedes*)

DESCRIPTION

The close function closes the file descriptor *filedes*.

EXAMPLE

```
int fd;

if (close(fd) != 0)
{
  /* handle close error conditions */
}
```

RETURNS

The normal return value from close is 0. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

ENODEV	The requested minor device does not exist.

This is the only error code returned by the driver, other codes may be returned by the I/O system during close. For more information about close error codes, see the *GNU C Library description – Low-Level Input/Output*.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.3 read()

NAME

read() – read from a device

SYNOPSIS

```
#include <unistd.h>

ssize_t read(int filedes, void *buffer, size_t size)
```

DESCRIPTION

The read function reads a CAN message from the specified receive queue. A pointer to the callers message buffer (*T903_MSG_BUF*) and the size of this structure are passed by the parameters *buffer* and *size* to the device.

The *T903_MSG_BUF* structure has the following layout:

```
typedef struct
{
        unsigned long           identifier;
        long                    timeout;
        unsigned char           rx_queue_num;
        unsigned char           extended;
        unsigned char           status;
        unsigned char           msg_len;
        unsigned char           data[8];
} T903_MSG_BUF, *PT903_MSG_BUF;
```

identifier

Receives the message identifier of the read CAN message.

timeout

Specifies the amount of time (in ticks) the caller is willing to wait for execution of read. A value of 0 means wait indefinitely.

rx_queue_num

Specifies the receive queue number from which the data will be read. Valid receive queue numbers are in range between 1 and n. In which n depends on the definition of *NUM_RX_QUEUES* (see also 2.6).

extended

Receives TRUE for extended CAN messages.

status

Receives status information about overrun conditions either in the CAN controller or intermediate software FIFO's.

T903_SUCCESS	No messages lost
T903_FIFO_OVERRUN	One or more messages was overwritten in the receive queue FIFO. This problem occurs if the FIFO is too small for the application read interval.
T903_MSGOBJ_OVERRUN	One or more messages were overwritten in the CAN controller message object because the interrupt latency is too large. Keep in mind Linux isn't a real-time operating system. Use message object 15 (buffered) to receive this time critical CAN messages, reduce the CAN bit rate or upgrade the system speed.
T903_RAW_FIFO_OVERRUN	One or more messages was overwritten in the FIFO between the interrupt service routine and post-processing in the driver (bottom half).

msg_len

Receives the number of message data bytes (0...8).

data

This buffer receives up to 8 data bytes. data[0] receives message data 0, data[1] receives message data 1 and so on.

EXAMPLE

```
#include       <tip903.h>

int            fd;
ssize_t        NumBytes;
T903_MSG_BUF   MsgBuf;

MsgBuf.rx_queue_num   = 1;
MsgBuf.timeout        = 200;

NumBytes = read(fd, &MsgBuf, sizeof(MsgBuf));
if (NumBytes > 0)
{
  /* process received CAN message */
}
```

RETURNS

On success read returns the size of structure T903_MSG_BUF. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL	Invalid argument. This error code is returned if either the size of the message buffer is too small, or the specified receive queue is out of range.
EFAULT	Invalid pointer to the message buffer.
ECONNREFUSED	The controller is in bus off state and no message is available in the specified receive queue. Note, as long as CAN messages are available in the receive queue FIFO, bus off conditions were not reported by a read function. This means you can read all CAN messages out of the receive queue FIFO during bus off state without an error result.
EAGAIN	Resource temporarily unavailable; the call might work if you try again later. This error occurs only if the device is opened with the flag O_NONBLOCK set.
ETIME	The allowed time to finish the read request is elapsed.
EINTR	Interrupted function call; an asynchronous signal occurred and prevented completion of the call. When this happens, you should try the call again.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.4 write()

NAME

write() – write to a device

SYNOPSIS

```
#include <unistd.h>

ssize_t write(int filedes, void *buffer, size_t size)
```

DESCRIPTION

The write function writes a CAN message to the device with descriptor *filedes*. A pointer to the callers message buffer (*T903_MSG_BUF*) and the size of this structure are passed by the parameters *buffer* and *size* to the device.

The write function dynamically allocates a free message object for the transmit operation. The search begins at message object 1 and ends at message object 14. The first found free message object is used. If currently no message object is available the write operation is blocked until any message object becomes free or a timeout occur.

If your application performs write operations you should left at least one message object free for transmit, preferably the first message object.

The *T903_MSG_BUF* structure has the following layout:

```
typedef struct
{
        unsigned long           identifier;
        long                    timeout;
        unsigned char           rx_queue_num;
        unsigned char           extended;
        unsigned char           status;
        unsigned char           msg_len;
        unsigned char           data[8];
} T903_MSG_BUF, *PT903_MSG_BUF;
```

identifier

Contains the message identifier of the CAN message to write.

timeout

Specifies the amount of time (in ticks) the caller is willing to wait for execution of write. A value of 0 means wait indefinitely.

rx_queue_num

This parameter is unused for this control function.

extended

This parameter is TRUE (1) for extended CAN messages.

status

This parameter is unused for this control function.

msg_len

Contains the number of message data bytes (0...8).

data

This buffer contains up to 8 data bytes. Data[0] contains message data 0, data[1] contains message data 1 and so on.

EXAMPLE

```
#include      <tip903.h>

int           fd;
ssize_t       NumBytes;
T903_MSG_BUF  MsgBuf;

MsgBuf.identifier  = 1234;
MsgBuf.timeout     = 200;
MsgBuf.extended    = TRUE;
MsgBuf.msg_len     = 2;
MsgBuf.data[0]     = 0xaa;
MsgBuf.data[1]     = 0x55;

NumBytes = write(fd, &MsgBuf, sizeof(MsgBuf));
if (NumBytes > 0)
{
  /* CAN message successful transmitted */
}
```

RETURNS

On success write returns the size of structure T903_MSG_BUF. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL	Invalid argument. This error code is returned if the size of the message buffer is too small.
EFAULT	Invalid pointer to the message buffer.
ECONNREFUSED	The controller is in bus off state and unable to transmit messages.
EAGAIN	Resource temporarily unavailable; the call might work if you try again later. This error occurs only if the device is opened with the flag *O_NONBLOCK* set.
ETIME	The allowed time to finish the write request is elapsed. This occurs if currently no message object is available or if the CAN bus is overloaded and the priority of the message identifier is too low.
EINTR	Interrupted function call; an asynchronous signal occurred and prevented completion of the call. When this happens, you should try the call again.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.5 ioctl()

NAME

ioctl() – device control functions

SYNOPSIS

#include <sys/ioctl.h>

int ioctl(int filedes, int request [, void *argp])

DESCRIPTION

The **ioctl** function sends a control code directly to a device, specified by *filedes*, causing the corresponding device to perform the requested operation.

The argument *request* specifies the control code for the operation. The optional argument *argp* depends on the selected request and is described for each request in detail later in this chapter.

The following ioctl codes are defined in *tip903.h*:

Symbol	Description
T903_IOCSBITTIMING	Setup new bit timing
T903_IOCSSETFILTER	Setup acceptance filter masks
T903_IOCGGETFILTER	Get the current acceptance filter masks
T903_IOCBUSON	Enter the bus on state
T903_IOCBUSOFF	Enter the bus off state
T903_IOCFLUSH	Flush one or all receive queues
T903_IOCGCANSTATUS	Returns the CAN controller status
T903_IOCSDEFRXBUF	Define a receive buffer message object
T903_IOCSDEFRMTBUF	Define a remote transmit buffer message object
T903_IOCSUPDATEBUF	Update a remote or receive buffer message object
T903_IOCTRELEASEBUF	Release an allocated message buffer object

See behind for more detailed information on each control code.

To use these TIP903 specific control codes the header file tip903.h must be included in the application

RETURNS

On success, zero is returned. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL	Invalid argument. This error code is returned if the requested ioctl function is unknown. Please check the argument *request*.

Other function dependant error codes will be described for each ioctl code separately. Note, the TIP903 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.5.1 T903_IOCSBITTIMING

NAME

T903_IOCSBITTIMING - Setup new bit timing

DESCRIPTION

This ioctl function modifies the bit timing register of the CAN controller to setup a new CAN bus transfer speed. A pointer to the callers parameter buffer (*T903_BITTIMING*) is passed by the argument *argp* to the driver.

Keep in mind to setup a valid bit timing value before changing into the Bus On state.

The *T903_BITTIMING* structure has the following layout:

```
typedef struct
{
    unsigned short          timing_value;
    unsigned short          three_samples;
} T903_BITTIMING, *PT903_BITTIMING;
```

timing_value

This parameter holds the new values for the bit timing register 0 (bit 0...7) and for the bit timing register 1 (bit 8...15). Possible transfer rates are between 20 kBit per second and 1.0 MBit per second. The include file 'tip903.h' contains predefined transfer rate symbols (T903_20KBIT ... T903_1_0MBIT).
For other transfer rates please follow the instructions of the Intel 82527 Architectural Overview, which is also part of the engineering kit TIP903-EK.

three_samples

If this parameter is TRUE (1) the CAN bus is sampled three times per bit time instead of one.

Use one sample point for faster bit rates and three sample points for slower bit rate to make the CAN bus more immune against noise spikes.

EXAMPLE

```
#include          <tip903.h>

int               fd;
int               result;
T903_BITTIMING    BitTimingParam;

BitTimingParam.timing_value = T903_100KBIT;
BitTimingParam.ThreeSamples = FALSE;
result = ioctl(fd, T903_IOCSBITTIMING, &BitTimingParam);

if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*

SEE ALSO

tip903.h for predefined bus timing constants

3.5.2 T903_IOCSSETFILTER

NAME

T903_IOCSSETFILTER - Setup acceptance filter masks

DESCRIPTION

This ioctl function modifies the acceptance filter masks of the specified CAN controller device.

The acceptance masks allow message objects to receive messages with a larger range of message identifiers instead of just a single message identifier. A "0" value means "don't care" or accept a "0" or "1" for that bit position. A "1"-value means that the incoming bit value "must-match" identically to the corresponding bit in the message identifier.

A pointer to the callers parameter buffer (*T903_ACCEPT_MASKS*) is passed by the parameter *argp* to the driver.

The *T903_ACCEPT_MASKS* structure has the following layout:

```
typedef struct
{
        unsigned long          message_15_mask;
        unsigned long          global_mask_extended;
        unsigned short         global_mask_standard;
} T903_ACCEPT_MASKS, *PT903_ACCEPT_MASKS;
```

message_15_mask

This parameter specifies the value for the Message 15 Mask Register. The Message 15 Mask Register is a local mask for message object 15. This 29 bit identifier mask appears in bit 3...31 of this parameter. The Message 15 Mask is "ANDed" with the Global Mask. This means that any bit defined as "don't care" in the Global Mask will automatically be a "don't care" bit for message 15. (See also Intel 82527 Architectural Overview).

global_mask_extended

This parameter specifies the value for the Global Mask-extended Register. The Global Mask-extended Register applies only to messages using the extended CAN identifier. This 29 bit identifier mask appears in bit 3...31 of this parameter.

global_mask_standard

This parameter specifies the value for the Global Mask-Standard Register. The Global Mask-Standard Register applies only to messages using the standard CAN identifier. The 11 bit identifier mask appears in bit 5...15 of this parameter.

The TIP903 device driver copies the parameter directly into the corresponding registers of the CAN controller, without shifting any bit positions. For more information see the Intel 82527 Architectural Overview.

EXAMPLE

```
#include          <tip903.h>

int               fd;
int               result;
T903_ACCEPT_MASKS AcceptMasksParam;

/* Standard identifier bits 0..3 don't care */
AcceptMasksParam.global_mask_standard = 0xfe00;

/* extended identifier bits 0..3 don't care */
AcceptMasksParam.global_mask_extended = 0xffffff80;

/* Message object 15 identifier bits 0..7 don't care */
AcceptMasksParam.message_15_mask = 0xfffff800;

result = ioctl(fd, T903_IOCSSETFILTER, &AcceptMasksParam);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.

3.5.3 T903_IOCGGETFILTER

NAME

T903_IOCGGETFILTER - Get the current acceptance filter masks

DESCRIPTION

This ioctl function returns the current acceptance filter masks of the specified CAN Controller.

A pointer to the callers parameter buffer (*T903_ACCEPT_MASKS*) is passed by the parameter *argp* to the driver.

The *T903_ACCEPT_MASKS* structure has the following layout:

```
typedef struct
{
        unsigned long           message_15_mask;
        unsigned long           global_mask_extended;
        unsigned short          global_mask_standard;
} T903_ACCEPT_MASKS, *PT903_ACCEPT_MASKS;
```

message_15_mask

This parameter receives the value for the Message 15 Mask Register. The Message 15 Mask Register is a local mask for message object 15. This 29 bit identifier mask appears in bit 3...31 of this parameter.

global_mask_extended

This parameter receives the value for the Global Mask-extended Register. The Global Mask-extended Register applies only to messages using the extended CAN identifier. This 29 bit identifier mask appears in bit 3...31 of this parameter.

global_mask_standard

This parameter receives the value for the Global Mask-Standard Register. The Global Mask-Standard Register applies only to messages using the standard CAN identifier. The 11 bit identifier mask appears in bit 5...15 of this parameter.

The TIP903 device driver copies the masks directly from the corresponding registers of the CAN controller into the parameter buffer, without shifting any bit positions. For more information see the Intel 82527 Architectural Overview.

EXAMPLE

```
#include           <tip903.h>

int                fd;
int                result;
T903_ACCEPT_MASKS  AcceptMasksParam;

result = ioctl(fd, T903_IOCGGETFILTER, &AcceptMasksParam);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.

3.5.4 T903_IOCBUSON

NAME

T903_IOCBUSON - Enter the bus on state

DESCRIPTION

This ioctl function sets the specified CAN controller into the Bus On state.

After an abnormal rate of occurrences of errors on the CAN bus or after driver startup, the CAN controller enters the Bus Off state. This control function resets the init bit in the control register. The CAN controller begins the busoff recovery sequence and resets transmit and receive error counters. If the CAN controller counts 128 packets of 11 consecutive recessive bits on the CAN bus, the Bus Off state is exited.

The optional argument can be omitted for this ioctl function.

Before the driver is able to communicate over the CAN bus after driver startup, this control function must be executed.

EXAMPLE

```
#include      <tip903.h>

int           fd;
int           result;

result = ioctl(fd, T903_IOCBUSON);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

This ioctl function returns no function specific error codes.

3.5.5 T903_IOCBUSOFF

NAME

T903_IOCBUSOFF - Enter the bus off state

DESCRIPTION

This ioctl function sets the specified CAN controller into the Bus-Off state. After a successful execution of this control function the CAN controller is completely removed from the CAN bus and cannot communicate until the control function T903_IOCBUSON is executed. Note: It’s not possible to set the device bus off during a write operation of another concurrent process.

The optional argument can be omitted for this ioctl function.

Execute this control function before the last close to the CAN controller channel.

EXAMPLE

```
#include       <tip903.h>

int            fd;
int            result;

result = ioctl(fd, T903_IOCBUSOFF);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EBUSY	Device busy. Another concurrent process is writing to the device at the moment. Try it again later.

3.5.6 T903_IOCFLUSH

NAME

T903_IOCFLUSH - Flush one or all receive queues

DESCRIPTION

This ioctl function flushes the message FIFO of the specified receive queue(s).

The optional argument *argp* passes the receive queue number to the device driver on which the FIFO's to be flushed. If this parameter is 0 the FIFO's of all receive queues of the device will be flushed, otherwise only the FIFO of the specified receive queue will be flushed.

EXAMPLE

```
#include      <tip903.h>

int           fd;
int           result;

/* flush all receive queues */

result = ioctl(fd, T903_IOCFLUSH, (int)0);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EINVAL	Invalid argument. This error code is returned if the specified receive queue is out of range.

3.5.7 T903_IOCGCANSTATUS

NAME

T903_IOCGCANSTATUS - Returns the CAN controller status

DESCRIPTION

This ioctl function returns the actual contents of the CAN controller status register for diagnostic purposes. A pointer to the callers status buffer (*T903_STATUS*) is passed by the parameter *argp* to the driver.

The *T903_STATUS* structure has the following layout:

```
typedef struct
{
        unsigned char           control_reg;
        unsigned char           status_reg;
} T903_STATUS, PT903_STATUS;
```

control_reg

This parameter receives the content of the controllers control register.

status_reg

This parameter receives the content of the controllers status register.

EXAMPLE

```
#include        <tip903.h>

int             fd, result;
T903_STATUS     CanStatus;

result = ioctl(fd, T903_IOCGCANSTATUS, &CanStatus);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer to the unsigned char variable which receives the contents of the CAN status register. Please check the argument *argp*.

3.5.8 T903_IOCSDEFRXBUF

NAME

T903_IOCSDEFRXBUF - Define a receive buffer message object

DESCRIPTION

This ioctl function defines a CAN message object to receive a single message identifier or a range of message identifiers (see also Acceptance Mask). All CAN messages received by this message object are directed to the associated receive queue and can be read with the standard read function (see also 3.3).

Before the driver can receive CAN messages it's necessary to define at least one receive message object. If only one receive message object is defined at all preferably message object 15 should be used because this message object is buffered.

A pointer to the caller's message description (*T903_BUF_DESC*) is passed by the argument *argp* to the driver.

The *T903_BUF_DESC* structure has the following layout:

```
typedef struct
{
        unsigned long           identifier;
        unsigned char           msg_obj_num;
        unsigned char           rx_queue_num;
        unsigned char           extended;
        unsigned char           msg_len;
        unsigned char           data[8];
} T903_BUF_DESC, *PT903_BUF_DESC;
```

identifier

This parameter specifies the message identifier for the message object to be defined.

msg_obj_num

Specifies the number of the message object to be defined. Valid object numbers are in range between 1 and 15.

rx_queue_num

Specifies the associated receive queue for this message object. All CAN messages received by this object are directed to this receive queue. The receive queue number is one based; valid numbers are in range between 1 and n. In which n depends on the definition of *NUM_RX_QUEUES* (see also 2.6).

extended

Set to TRUE for extended CAN messages.

msg_len

This parameter is unused for this control function.

data

This parameter is unused for this control function.

It’s possible to assign more than one receive message object to one receive queue.

EXAMPLE

```
#include        <tip903.h>

int             fd;
int             result;
T903_BUF_DESC   BufDesc;

BufDesc.msg_obj_num  = 15;
BufDesc.rx_queue_num = 1;
BufDesc.identifier = 1234;
BufDesc.extended   = TRUE;

/* Define message object 15 to receive the extended         */
/* message identifier 1234 and store received messages      */
/* in receive queue 1                                       */

result = ioctl(fd, T903_IOCSDEFRXBUF, &BufDesc);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.
EINVAL	Invalid argument. This error code is returned if either the message object number, or the specified receive queue is out of range.
EADDRINUSE	The requested message object is already occupied.

3.5.9 T903_IOCSDEFRMTBUF

NAME

T903_IOCSDEFRMTBUF - Define a remote transmit buffer message object

DESCRIPTION

This ioctl function defines a remote transmission CAN message buffer object. A remote transmission object is similar to normal transmission object with exception that the CAN message is transmitted only after receiving of a remote frame with the same identifier.

This type of message object can be used to make process data available for other nodes which can be polled around the CAN bus without any action of the provider node.

The message data remain available for other CAN nodes until this message object is updated with the control function *T903_IOCSUPDATEBUF* or cancelled with *T903_IOCTRELEASEBUF*.

A pointer to the caller's message description (*T903_BUF_DESC*) is passed by the argument *argp* to the driver.

The *T903_BUF_DESC* structure has the following layout:

```
typedef struct
{
        unsigned long           identifier;
        unsigned char           msg_obj_num;
        unsigned char           rx_queue_num;
        unsigned char           extended;
        unsigned char           msg_len;
        unsigned char           data[8];
} T903_BUF_DESC, *PT903_BUF_DESC;
```

identifier

This parameter specifies the message identifier for the message object to be defined.

msg_obj_num

Specifies the number of the message object to be defined. Valid object numbers are in range between 1 and 14.

Keep in mind that message object 15 is only available for receive message objects.

rx_queue_num

Unused for remote transmission message objects. Set to 0.

extended

Set to TRUE for extended CAN messages.

msg_len

Contains the number of message data bytes (0...8).

data

This buffer contains up to 8 data bytes. data[0] contains message data 0, data[1] contains message data 1 and so on.

EXAMPLE

```
#include        <tip903.h>

int             fd;
int             result;
T903_BUF_DESC   BufDesc;

BufDesc.msg_obj_num  = 10;
BufDesc.identifier   = 777;
BufDesc.extended     = TRUE;
BufDesc.msg_len      = 1;
BufDesc.data[0]      = 123;

/* Define message object 10 to transmit the extended  */
/* message identifier 777 after receiving of a remote */
/* frame with the same identifier                     */

result = ioctl(fd, T903_IOCSDEFRMTBUF, &BufDesc);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.
EINVAL	Invalid argument. This error code is returned if the message object number is out of range.
EADDRINUSE	The requested message object is already occupied.
EMSGSIZE	Invalid message size. msg_len must be in range between 0 and 8.

3.5.10 T903_IOCSUPDATEBUF

NAME

T903_IOCSUPDATEBUF - Update a remote or receive buffer message object

DESCRIPTION

This ioctl function updates a previous defined receive or remote transmission message buffer object.

To update a receive message object a remote frame is transmitted over the CAN bus to request new data from a corresponding remote transmission message object on other nodes.

To update a remote transmission object only the message data and message length of the specified message object is changed. No transmission is initiated by this control function.

A pointer to the caller's message description (*T903_BUF_DESC*) is passed by the argument *argp* to the driver.

The *T903_BUF_DESC* structure has the following layout:

```
typedef struct
{
        unsigned long           identifier;
        unsigned char           msg_obj_num;
        unsigned char           rx_queue_num;
        unsigned char           extended;
        unsigned char           msg_len;
        unsigned char           data[8];
} T903_BUF_DESC, *PT903_BUF_DESC;
```

identifier

This parameter is unused for this control function.

msg_obj_num

Specifies the number of the message object to be updated. Valid object numbers are in range between 1 and 14.

Keep in mind that message object 15 is available only for receive message objects.

rx_queue_num

This parameter is unused for this control function.

extended

Set to TRUE for extended CAN messages.

msg_len

Contains the number of message data bytes (0...8). This parameter is used only for remote transmission object updates.

data

This buffer contains up to 8 data bytes. data[0] contains message data 0, data[1] contains message data 1 and so on.

This parameter is used only for remote transmission object updates.

EXAMPLE

```
#include        <tip903.h>

int             fd;
int             result;
T903_BUF_DESC   BufDesc;

/* Update a receive message object */
BufDesc.msg_obj_num  = 14;
result = ioctl(fd, T903_IOCSUPDATEBUF, &BufDesc);
if (result < 0)
{
  /* handle ioctl error */
}

/* Update a remote message object */
BufDesc.msg_obj_num  = 10;
BufDesc.msg_len      = 1;
BufDesc.data[0]      = 124;
result = ioctl(fd, T903_IOCSUPDATEBUF, &BufDesc);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.
EINVAL	Invalid argument. This error code is returned if either the message object number is out of range or the requested message object is not defined.
EMSGSIZE	Invalid message size. *msg_len* must be in range between 0 and 8.

3.5.11 T903_IOCTRELEASEBUF

NAME

T903_IOCTRELEASEBUF - Release an allocated message buffer object

DESCRIPTION

This TIP903 control function releases a previous defined CAN message object. Any CAN bus transactions of the specified message object become disabled. After releasing the message object can be defined again with *T903_IOCSDEFRXBUF* and *T903_IOCSDEFRMTBUF* control functions.

A pointer to the caller's message description (*T903_BUF_DESC*) is passed by the argument *argp* to the driver.

The *T903_BUF_DESC* structure has the following layout:

```
typedef struct
{
        unsigned long           identifier;
        unsigned char           msg_obj_num;
        unsigned char           rx_queue_num;
        unsigned char           extended;
        unsigned char           msg_len;
        unsigned char           data[8];
} T903_BUF_DESC, *PT903_BUF_DESC;
```

msg_obj_num

Specifies the number of the message object to be released. Valid object numbers are in range between 1 and 15.

All other parameters are not used and could be left blank.

EXAMPLE

```
#include         <tip903.h>

int              fd;
int              result;
T903_BUF_DESC    BufDesc;

BufDesc.msg_obj_num  = 14;

result = ioctl(fd, T903_IOCTRELEASEBUF, &BufDesc);
if (result < 0)
{
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Invalid pointer to the parameter buffer. Please check the argument *argp*.
EINVAL	Invalid argument. This error code is returned if the message object number is out of range.
EBADMSG	The requested message object is not defined.
EBUSY	The message object is currently busy transmitting data.

4 Debugging

For debugging output see tip903drv.c. You will find the two following symbols:

```
#undef TIP903_DEBUG_INTR
#undef TIP903_DEBUG_VIEW
```

To enable a debug output replace “undef” with “define”.

The TIP903_DEBUG_INTR symbol controls debugging output from the ISR.

```
TIP903  :  interrupt entry
TIP903  :  IACK[0] vector = 0005
```

The TIP903_DEBUG_VIEW symbol controls debugging output from the remaining part of the driver.

```
TIP903 - 3 Channel extended CAN Bus IP - version 1.2.3 (2010-11-25)
TIP903  :  Probe new TIP903 mounted on <TEWS TECHNOLOGIES - (Compact)PCI
IPAC Carrier> at slot A
TIP903  :  IP MEM Memory Space
00000000 : 01 00 61 61 01 00 FF FF FF FF FF F8 00 00 00 00
00000010 : 95 55 74 A0 00 20 00 00 00 00 00 00 00 00 00 00
00000020 : 56 55 02 00 24 00 00 80 40 00 00 00 00 00 40 00
00000030 : 56 55 00 05 04 00 00 00 00 00 02 00 00 00 00 00
00000040 : 55 55 00 34 95 40 00 00 00 00 00 00 00 00 00 00
00000050 : 56 55 00 00 00 00 08 00 80 00 00 82 00 00 00 00
00000060 : 55 55 00 10 00 00 40 00 08 00 00 02 00 00 00 FF
00000070 : 95 59 02 00 02 00 00 00 40 00 20 00 00 00 00 FF
00000080 : 55 55 12 00 02 10 00 00 40 00 00 00 00 42 40 FF
00000090 : 55 55 00 80 20 50 00 00 00 00 00 00 20 00 80 00
000000A0 : 55 55 08 08 00 00 08 04 00 00 00 00 00 00 00 00
000000B0 : 55 55 32 01 04 00 00 00 00 00 00 00 90 00 00 00
000000C0 : 55 65 61 34 10 40 08 00 00 00 00 04 80 00 00 FF
000000D0 : 55 55 A8 62 20 00 00 00 08 00 00 00 00 01 00 00
000000E0 : 55 69 00 34 00 40 04 00 00 00 00 00 40 00 00 00
000000F0 : 95 55 1D A4 07 20 00 00 00 04 10 00 00 00 00 FF

TIP903  :  IP MEM Memory Space
00000100 : 01 00 61 61 01 00 FF FF FF FF FF F8 00 00 00 00
...
000001F0 : 95 55 40 53 D1 18 40 00 00 00 00 00 00 00 00 FF

TIP903  :  IP MEM Memory Space
00000200 : 01 00 61 61 01 00 FF FF FF FF FF F8 00 00 00 00
...
000002F0 : 95 55 8E 96 AC 48 00 00 00 20 20 00 00 80 10 FF
```

If you have trouble with the driver please enable the debug output and send us a copy of the results. The kernel context output is generated with “printk” and is appended to /var/log/messages or wherever it’s piped in your system.

For debugging please run

```
# tail –f /var/log/messages
```

at first and then install the driver.