The Embedded I/O Company

TEWS TECHNOLOGIES

TIP903-SW-72

LynxOS Device Driver

Three Channel Extended CAN Bus IP

Version 2.1.x

User Manual

Issue 2.1.0

March 2009

TEWS TECHNOLOGIES GmbH
Am Bahnhof 7
25469 Halstenbek, Germany
www.tews.com
Phone: +49 (0) 4101 4058 0
Fax: +49 (0) 4101 4058 19
e-mail: info@tews.com

TEWS TECHNOLOGIES LLC
9190 Double Diamond Parkway,
Suite 127, Reno, NV 89521, USA
www.tews.com
Phone: +1 (775) 850 5830
Fax: +1 (775) 201 0347
e-mail: usasales@tews.com

TIP903-SW-72

LynxOS Device Driver

Three Channel Extended CAN Bus IP

Supported Modules:
TIP903

Issue	Description	Date
1.0	First Issue	June 11, 2003
2.0	IPAC Carrier Device Driver	December 7, 2003
2.1.0	Revision	March 6, 2009

Table of Contents

1 Introduction

The TIP903-SW-72 LynxOS device driver allows the operation of a TIP903 Extended CANbus IP on LynxOS operating systems.

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

The standard file (I/O) functions (open, close, read, write and ioctl) provide the basic interface for opening and closing a file descriptor and for performing device I/O and control operations.

The TIP903 device driver includes the following functions:

- 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
- 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
- TEWS TECHNOLOGIES IPAC carrier driver support.

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

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

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-72 IPAC Carrier User Manual

1.1 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-72 is part of this TIP903-SW-72 distribution. It is located in directory CARRIER-SW-72 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-72 User Manual for a detailed description how to install and setup the CARRIER-SW-72 device driver, and for a description of the TEWS TECHNOLOGIES IPAC Carrier Driver concept.

2 Installation

Following files are located on the distribution media:

Directory path '.\TIP903-SW-72\':

TIP903-SW-72-SRC.tar.gz	GZIP compressed archive with driver source code
TIP903-SW-72-2.1.0.pdf	PDF copy of this manual
ChangeLog.txt	Release history
Release.txt	Release information

For installation the files have to be copied to the desired target directory.

The GZIP compressed archive TIP903-SW-72-SRC.tar.gz contains the following files and directories:

Directory path './tip903/':

tip903.c	Driver source code
tip903.h	Definitions and data structures for driver and application
tip903def.h	Definitions and data structures for the driver
i82527.h	Extended CAN controller programming model
tip903_info.c	Device information definition
tip903_info.h	Device information definition header
tip903.cfg	Driver configuration file include
tip903.import	Linker import file for PowerPC platforms
Makefile	Device driver make file
Example/tip903exa.c	Example application source
Example/Makefile	Example make file

In order to perform an installation, extract all files of the archive TIP903-SW-72-SRC.tar.gz to the desired target directory. The command 'tar -xzvf TIP903-SW-72-SRC.tar.gz' will extract the files into the local directory.

(1) Create a new directory in the system driver directory path /sys/drivers.xxx, where xxx represents the BSP that supports the target hardware.

For example: /sys/drivers.pp_drm/tip903 or /sys/drivers.cpci_x86/tip903

(2) Copy the following files to this directory:
- tip903.c
- tip903def.h
- i82527.h
- tip903.import
- Makefile

(3) Copy tip903.h to /usr/include/

(4) Copy tip903_info.c to /sys/devices.xxx/ or /sys/devices if /sys/devices.xxx does not exist (xxx represents the BSP).

(5) Copy tip903_info.h to /sys/dheaders/

(6) Copy tip903.cfg to */sys/cfg.xxx/*, where xxx represents the BSP for the target platform. For example: /sys/cfg.ppc or /sys/cfg.x86 ....

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 *A:\CARRIER-SW-72* on the separate distribution diskette.

2.1 Device Driver Installation

The two methods of driver installation are as follows:

- Static Installation
- Dynamic Installation (only native LynxOS systems)

Both installation methods require the TEWS TECHNOLOGIES IPAC Carrier Driver. Please refer to the IPAC Carrier Driver User Manual for detailed information.

2.1.1 Static Installation

With this method, the driver object code is linked with the kernel routines and is installed during system start-up.

2.1.1.1 Build the driver object

(1) Change to the directory /sys/drivers.xxx/tip903, where xxx represents the BSP that supports the target hardware.

(2) To update the library /sys/lib/libdrivers.a enter:

```
make install
```

2.1.1.2 Create Device Information Declaration

(1) Change to the directory /sys/devices.xxx/ or /sys/devices if /sys/devices.xxx does not exist (xxx represents the BSP).

(2) Add the following dependencies to the Makefile

```
DEVICE_FILES_all = ... tip903_info.x
```

And at the end of the Makefile

```
tip903_info.o:$(DHEADERS)/tip903_info.h
```

(3) To update the library /sys/lib/libdevices.a enter:

```
make install
```

2.1.1.3 Modify the Device and Driver Configuration File

In order to insert the driver object code into the kernel image, an appropriate entry in file CONFIG.TBL must be created.

(1) Change to the directory /sys/lynx.os/ respective /sys/bsp.xxx, where xxx represents the BSP that supports the target hardware.

(2) Create an entry at the end of the file CONFIG.TBL

Insert the following entry at the end of this file. Be sure that the necessary TEWS TECHNOLOGIES IPAC carrier driver is included **before** this entry.

```
I:tip903.cfg
```

2.1.1.4 Rebuild the Kernel

(1) Change to the directory /sys/lynx.os/ (/sys/bsp.xxx)

(2) Enter the following command to rebuild the kernel:

```
make install
```

(3) Reboot the newly created operating system by the following command (not necessary for KDIs):

```
reboot –aN
```

The N flag instructs init to run mknod and create all the nodes mentioned in the new nodetab.

(4) After reboot you should find the following new devices (depends on the device configuration): /dev/tip903_0, /dev/tip903_1, /dev/tip903_2, …

2.1.2 Dynamic Installation

This method allows you to install the driver after the operating system is booted. The driver object code is attached to the end of the kernel image and the operating system dynamically adds this driver to its internal structures. The driver can also be removed dynamically.

2.1.2.1 Build the driver object

(1) Change to the directory /sys/drivers.xxx/tip903, where xxx represents the BSP that supports the target hardware.

(2) To make the dynamic link-able driver enter :

```
make
```

2.1.2.2 Create Device Information Declaration

(1) Change to the directory /sys/drivers.xxx/tip903, where xxx represents the BSP that supports the target hardware.

(2) To create a device definition file for the major device (this works only on native system)

```
make t903info
```

(3) To install the driver enter:

```
drinstall –c tip903.obj
```

If successful, drinstall returns a unique <driver-ID>

(4) To install the major device enter:

```
devinstall –c –d <driver-ID> t903info
```

The <driver-ID> is returned by the drinstall command

(5) To create nodes for the devices enter:

```
mknod /dev/tip903_0 c <major_no> 0
mknod /dev/tip903_1 c <major_no> 1
mknod /dev/tip903_2 c <major_no> 2
...
```

The <major_no> is returned by the devinstall command.

If all steps are successful completed the TIP903 is ready to use.

2.1.2.3 Uninstall dynamic loaded driver

To uninstall the TIP903 device enter the following commands:

```
devinstall –u –c <device-ID>
drinstall –u <driver-ID>
```

2.1.3 Configuration File: CONFIG.TBL

The device and driver configuration file CONFIG.TBL contains entries for device drivers and its major and minor device declarations. Each time the system is rebuild, the config utility read this file and produces a new set of driver and device configuration tables and a corresponding nodetab.

To install the TIP903 driver and devices into the LynxOS system, the configuration include file tip903.cfg must be included in the CONFIG.TBL (see also 2.1.1.3).

The file tip903.cfg on the distribution disk contains the driver entry (*C:tip903:\....*) and a major device entry (*D:TIP903:t903info::*) with 9 minor device entries (*"N: tip903_0:0", ..., "N: tip903_8:8"*).

If the driver should support more than 9 minor devices (CAN channels) because more than 3 TIP903 are plugged, additional minor device entries must be added. To create the device node */dev/tip903_9* the line *N:tip903_9:9* must be added at the end of the file tip903.cfg. For the next node a minor device entry with 10 must be added and so on.

This example shows the predefined driver entry:

```
#    Format:
#    C:driver-name:open:close:read:write:select:control:install:uninstall
#    D:device-name:info-block-name:raw-partner-name
#    N:node-name:minor-dev

C:tip903:\
     :t903open:t903close:t903read:t903write:\
     ::t903ioctl:t903install:t903uninstall
D:TIP903:t903info::
N:tip903_0:0
N:tip903_1:1
N:tip903_2:2
N:tip903_3:3
N:tip903_4:4
N:tip903_5:5
N:tip903_6:6
N:tip903_7:7
N:tip903_8:8
```

The configuration above creates the following node in the /dev directory.

/dev/tip903_0 ... /dev/tip903_8

2.2 Receive Queue Configuration

Received CAN messages will be stored in receive queues. Each receive queue contains a FIFO. The number of receive queues and the depth of the FIFO can be adapted by changing the following symbols in tip903def.h.

NUM_RX_QUEUES

Defines the number of receive queues for each device (default = 3). Valid numbers are in range between 1 and 15.

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 I/O Functions

LynxOS system calls are all available directly to any C program. They are implemented as ordinary function calls to "glue" routines in the system library, which trap to the OS code.

Note that many system calls use data structures, which should be obtained in a program from appropriate header files. Necessary header files are listed with the system call synopsis.

3.1 open()

NAME

open() - open a file

SYNOPSIS

#include <sys/file.h>
#include <sys/types.h>
#include <fcntl.h>

int open (char *path, int oflags[, mode_t mode])

DESCRIPTION

Opens a file (TIP903 device) named in path for reading and writing. The value of oflags indicates the intended use of the file. In case of a TIP903 devices oflags must be set to O_RDWR to open the file for both reading and writing.

The mode argument is required only when a file is created. Because a TIP903 device already exists this argument is ignored.

EXAMPLE

```
int  fd

fd = open ("/dev/tip903_0", O_RDWR);
```

RETURNS

Open returns a file descriptor number if successful or 1 on error. The global variable *errno* contains the detailed error code.

3.2 close()

NAME

close() – close a file

SYNOPSIS

int close(int fd)

DESCRIPTION

This function closes an opened device associated with the valid file descriptor handle fd.

EXAMPLE

```
int  result;

result = close(fd);
```

RETURNS

Close returns 0 (OK) if successful, or –1 on error. The global variable errno contains the detailed error code.

SEE ALSO

LynxOS System Call - close()

3.3 read()

NAME

read() - read from a file

SYNOPSIS

#include <tip903.h>

int read (int fd, char *buff, int count)

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 buff and count to the device.

```
typedef struct {
        unsigned long     Identifier;
        long              Timeout;
        unsigned char     RxQueueNum;
        unsigned char     Extended;
        unsigned char     Status;
        unsigned char     MsgLen;
        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.

RxQueueNum

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 2.2).

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. Reduce the CAN bit rate or upgrade the system speed.

MsgLen

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

```
int  fd;
int  result;
T903_MSG_BUF  MsgBuf;

MsgBuf.RxQueueNum  = 1;
MsgBuf.Timeout     = 200;

result = read(fd, (char*)&MsgBuf, sizeof(MsgBuf));

if(result != sizeof(T903_MSG_BUF)) {
     /* handle read error */
}
```

RETURNS

When read succeeds, the size of the read buffer is returned. If read fails, -1 (SYSERR) is returned.

On error, errno will contain a standard read error code (see also LynxOS System Call – read) or one of the following TIP903 specific error codes:

ENXIO	Illegal device
EINVAL	Invalid argument. This error code is returned if either the size of the message buffer is too small, or the specified receive queue number is out of range.
ETIMEDOUT	The maximum allowed time to finish the read request is exhausted.
ENETDOWN	The controller is in bus off state and no message is available in the specified receive queue.
EAGAIN	You've set a timeout value, but there are no timeout timer available. Do it again without a timeout.
EINTR	Interrupted system call (probably by a signal).

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.

SEE ALSO

LynxOS System Call - read()

3.4 write()

NAME

write() – write to a file

SYNOPSIS

int write (int fd, char *buff, int count)

DESCRIPTION

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

The write function dynamically allocates a free message object for this transmit operation. The search begins at message object 1 and ends at message object 14. The first found free message object is used. If no message object is available the write operation returns with error.

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

```
typedef struct {
        unsigned long     Identifier;
        long              Timeout;
        unsigned char     RxQueueNum;
        unsigned char     Extended;
        unsigned char     Satus;
        unsigned char     MsgLen;
        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.

RxQueueNum

Unused for this control function. Can be 0.

Extended

Contains TRUE (1) for extended CAN messages.

Status

Unused for this control function. Can be 0.

MsgLen

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

```
int  fd;
int  result;
T903_MSG_BUF  MsgBuf;

/*
**   Write two data bytes with identifier 1234 to the
**   CANbus and wait max. 200 ticks on execution
*/

MsgBuf.Identifier  = 1234;
MsgBuf.Timeout     = 200;
MsgBuf.Extended    = TRUE;
MsgBuf.MsgLen      = 2;
MsgBuf.Data[0]     = 0xaa;
MsgBuf.Data[1]     = 0x55;

result = write(fd, &MsgBuf, sizeof(MsgBuf));

if(result != sizeof(T903_MSG_BUF)) {
     /* handle write error */
}
```

RETURNS

When write succeeds, the size of the write buffer is returned. If write fails, -1 (SYSERR) is returned.

On error, errno will contain a standard write error code (see also LynxOS System Call – write) or the following TIP903 specific error code:

ENXIO	Illegal device
EINVAL	Invalid argument. This error code is returned if the size of the message buffer is too small.
ENOSPC	No free message object available for transmit.
ENETDOWN	The controller is in bus off state and unable to transmit messages.
ETIMEDOUT	The allowed time to finish the write request is elapsed. This occurs if the CAN bus is overloaded and the priority of the message identifier is too low, no other node is online or the controller enters the BusOff state.
EAGAIN	You've set a timeout value, but there are no timeout timer available. Do it again without a timeout.
EINTR	Interrupted system call (probably by a signal).

SEE ALSO

LynxOS System Call - write()

3.5 ioctl()

NAME

ioctl() - I/O device control

SYNOPSIS

#include <ioctl.h>
#include <tip903.h>

int ioctl (int fd, int request, char *arg)

DESCRIPTION

ioctl provides a way of sending special commands to a device driver. The call sends the value of request and the pointer arg to the device associated with the descriptor fd.

The following ioctl codes are defined in tip903.h :

Value	Meaning
T903_BITTIMING	Setup a new bit timing
T903_SETFILTER	Setup acceptance filter masks
T903_GETFILTER	Get the current acceptance filter masks
T903_BUSON	Enter the bus on state
T903_BUSOFF	Enter the bus off state
T903_FLUSH	Flush one or all receive queues
T903_CANSTATUS	Returns the contents of the CAN controller status register
T903_DEFRXBUF	Define a receive buffer message object
T903_DEFRMTBUF	Define a remote transmit buffer message object
T903_UPDATEBUF	Update a remote or receive buffer message object
T903_RELEASEBUF	Release an allocated message buffer object

See behind for more detailed information on each control code.

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. The TIP903 ioctl function returns always standard error codes.

SEE ALSO

LynxOS System Call – ioctl() for detailed description of possible error codes.

3.5.1 T903_BITTIMING

NAME

T903_BITTIMING - 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 caller's parameter buffer (T903_TIMING) is passed by the argument pointer arg to the driver.

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

```
typedef struct {
        unsigned short      TimingValue;
        unsigned short      ThreeSamples;
} T903_TIMING, *PT903_TIMING;
```

TimingValue

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_1MBIT).

For other transfer rates please follow the instructions of the Intel 82527 Architectural Overview, which is also part of the engineering kit TIP903-EK.

ThreeSamples

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

NOTE: 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

```
int  fd;
int  result;
T903_TIMING  BitTimingParam;

BitTimingParam.TimingValue  = T903_100KBIT;
BitTimingParam.ThreeSamples = FALSE;

result = ioctl(fd, T903_TIMING, (char*)&BitTimingParam);

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

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

tip903.h for predefined bus timing constants

Intel 82527 Architectural Overview - 4.13 Bit Timing Overview

3.5.2 T903_SETFILTER

NAME

T903_SETFILTER - 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 caller's parameter buffer (T903_ACCEPT_MASKS) is passed by the parameter pointer arg to the driver.

```
typedef struct {
        unsigned long       Message15Mask;
        unsigned long       GlobalMaskExtended;
        unsigned short      GlobalMaskStandard;
} T903_ACCEPT_MASKS, *PT903_ACCEPT_MASKS;
```

Message15Mask

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 "AND'ed" 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).

GlobalMaskExtended

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.

GlobalMaskStandard

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.

NOTE: 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 - 4.7…4.10

EXAMPLE

```
int  fd;
int  result;
T903_ACCEPT_MASKS  AcceptMasksParam;

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

/* Extended identifier bits 0..3 don't care */
AcceptMasksParam.GlobalMaskExtended = 0xffffff80;

/* Message object 15 identifier bits 0..7 don't care */
AcceptMasksParam.Message15Mask = 0xfffff800;

result = ioctl(fd, T903_SETFILTER, (char*)&AcceptMasksParam);

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

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

Intel 82527 Architectural Overview - 4.9 Acceptance Filtering

3.5.3 T903_GETFILTER

NAME

T903_GETFILTER - 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 caller's parameter buffer (T903_ACCEPT_MASKS) is passed by the parameter pointer arg to the driver.

```
typedef struct {
        unsigned long       Message15Mask;
        unsigned long       GlobalMaskExtended;
        unsigned short      GlobalMaskStandard;
} T903_ACCEPT_MASKS, *PT903_ACCEPT_MASKS;
```

Message15Mask

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.

GlobalMaskExtended

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.

GlobalMaskStandard

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.

NOTE: 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 - 4.7…4.10

EXAMPLE

```
int  fd;
int  result;
T903_ACCEPT_MASKS  AcceptMasksParam;


result = ioctl(fd, T903_GETFILTER, (char*)&AcceptMasksParam);

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

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

Intel 82527 Architectural Overview - 4.9 Acceptance Filtering

3.5.4 T903_BUSON

NAME

T903_BUSON - 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 the 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 pointer can be NULL.

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

EXAMPLE

```
int  fd;
int  result;


result = ioctl(fd, T903_BUSON, NULL);

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

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

Intel 82527 Architectural Overview - 3.2 Software Initialization

3.5.5 T903_BUSOFF

NAME

T903_BUSOFF - Enter the bus off state

DESCRIPTION

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

After execution of this control function the CAN controller is completely removed from the CAN bus and cannot communicate until the control function T903_BUSON is executed.

The optional argument pointer can be NULL.

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

EXAMPLE

```
int  fd;
int  result;


result = ioctl(fd, T903_BUSOFF, NULL);

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

ERRORS

This ioctl function returns no function specific error codes.

SEE ALSO

Intel 82527 Architectural Overview - 3.2 Software Initialization

3.5.6 T903_FLUSH

NAME

T903_FLUSH - Flush one or all receive queues

DESCRIPTION

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

The optional argument pointer arg passes the receive queue number to the device driver on which the FIFO's to be flushed.

EXAMPLE

```
int  fd;
int  result;
char  RxQueueNum;


/* flush receive queues 1 */
RxQueueNum = 1;

result = ioctl(fd, T903_IOCFLUSH, &RxQueueNum);

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

ERRORS

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

3.5.7 T903_DEFRXBUF

NAME

T903_DEFRXBUF - 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 double-buffered.

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

```
typedef struct {
        unsigned long     Identifier;
        unsigned char     MsgObjNum;
        unsigned char     RxQueueNum;
        unsigned char     Extended;
        unsigned char     MsgLen;
        unsigned char     Data[8];
} T903_BUF_DESC, *PT903_BUF_DESC;
```

Identifier

Specifies the message identifier for the message object to be defined.

MsgObjNum

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

RxQueueNum

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 2.2).

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

Extended

Set to TRUE for extended CAN messages.

MsgLen

Unused for this control function. Set to 0.

Data

Unused for this control function.

EXAMPLE

```
int  fd;
int  result;
T903_BUF_DESC  BufDesc;


BufDesc.MsgObjNum  = 15;
BufDesc.RxQueueNum = 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_DEFRXBUF, (char*)&BufDesc);

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

ERRORS

EINVAL	Invalid argument. This error code is returned if either the message object number or the specified receive queue number is out of range.
EADDRINUSE	The requested message object is already occupied.

SEE ALSO

Intel 82527 Architectural Overview - 4.18 82527 Message Objects

3.5.8 T903_DEFRMTBUF

NAME

T903_DEFRMTBUF - 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_UPDATEBUF or cancelled with T903_RELEASEBUF.

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

```
typedef struct {
        unsigned long       Identifier;
        unsigned char       MsgObjNum;
        unsigned char       RxQueueNum;
        unsigned char       Extended;
        unsigned char       MsgLen;
        unsigned char       Data[8];
} T903_BUF_DESC, *PT903_BUF_DESC;
```

Identifier

Specifies the message identifier for the message object to be defined.

MsgObjNum

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.

RxQueueNum

Unused for remote transmission message objects. Set to 0.

Extended

Set to TRUE for extended CAN messages.

MsgLen

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

Data[8]

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

EXAMPLE

```
int  fd;
int  result;
T903_BUF_DESC  BufDesc;


BufDesc.MsgObjNum  = 10;
BufDesc.Identifier = 777;
BufDesc.Extended   = TRUE;
BufDesc.MsgLen     = 1;
BufDesc.Data[0]    = 123;

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

result = ioctl(fd, T903_DEFRMTBUF, (char*)&BufDesc);

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

ERRORS

EINVAL	Invalid argument. This error code is returned if the message object number or the message length (MsgLen) is out of range.
EADDRINUSE	The requested message object is already occupied.

SEE ALSO

Intel 82527 Architectural Overview - 4.18 82527 Message Objects

3.5.9 T903_UPDATEBUF

NAME

T903_UPDATEBUF - 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 pointer arg to the driver.

```
typedef struct {
        unsigned long     Identifier;
        unsigned char     MsgObjNum;
        unsigned char     RxQueueNum;
        unsigned char     Extended;
        unsigned char     MsgLen;
        unsigned char     Data[8];
} T903_BUF_DESC, *PT903_BUF_DESC;
```

Identifier

Unused for this control function. Set to 0.

MsgObjNum

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

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

RxQueueNum

Unused for this control function. Set to 0.

Extended

Set to TRUE for extended CAN messages.

MsgLen

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

```
int  fd;
int  result;
T903_BUF_DESC  BufDesc;


/* Update a receive message object */
BufDesc.MsgObjNum  = 14;

result = ioctl(fd, T903_UPDATEBUF, (char*)&BufDesc);

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

/* Update a remote message object */
BufDesc.MsgObjNum  = 10;
BufDesc.MsgLen     = 1;
BufDesc.Data[0]    = 124;

result = ioctl(fd, T903_UPDATEBUF, (char*)&BufDesc);

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

ERRORS

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. MsgLen must be in range between 0 and 8.

SEE ALSO

Intel 82527 Architectural Overview - 4.18 82527 Message Objects

3.5.10 T903_RELEASEBUF

NAME

T903_RELEASEBUF - 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_DEFRXBUF and T903_DEFRMTBUF control functions.

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

```
typedef struct {
        unsigned long           Identifier;
        unsigned char           MsgObjNum;
        unsigned char           RxQueueNum;
        unsigned char           Extended;
        unsigned char           MsgLen;
        unsigned char           Data[8];
} T903_BUF_DESC, *PT903_BUF_DESC;
```

MsgObjNum

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

```
int  fd;
int  result;
T903_BUF_DESC  BufDesc;


BufDesc.MsgObjNum  = 14;

result = ioctl(fd, T903_RELEASEBUF, (char*)&BufDesc);

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

ERRORS

EINVAL	Invalid argument. This error code is returned if the message object number is out of range or the requested message object is not defined.
EBUSY	The message object is currently busy transmitting data or other tasks are waiting for a received message.

SEE ALSO

ioctl man pages

3.5.11 T903_CANSTATUS

NAME

T903_CANSTATUS - Returns the contents of the CAN status register

DESCRIPTION

This ioctl function returns the actual contents of the CAN controller status register for diagnostic purposes.

The contents of the controller status register is received in an unsigned char variable. A pointer to this variable is passed by the argument pointer arg to the driver.

EXAMPLE

```
int  fd;
int  result;
unsigned char  CanStatus;


result = ioctl(fd, T903_CANSTATUS, (char*)&CanStatus);

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

SEE ALSO

Intel 82527 Architectural Overview - 4.3 Status Register

3.6 Step by Step Driver Initialization

The following code example illustrates all necessary steps to initialize a CAN device for communication.

```
/*
**  ( 1.) Setup CAN bus bit timing
*/
BitTimingParam.TimingValue   = T903_100KBIT;
BitTimingParam.ThreeSamples  = FALSE;

result = ioctl(fd, T903_TIMING, (char*)&BitTimingParam);


/*
**  ( 2.) Setup acceptance filter masks
*/
AcceptMasksParam.GlobalMaskStandard  = 0xFFFF;
AcceptMasksParam.GlobalMaskExtended  = 0xFFFFFFFF;
AcceptMasksParam.Message15Mask       = 0;

result = ioctl(fd, T903_SETFILTER, (char*)&AcceptMasksParam);


/*
**  ( 3.) Define message object 15 for reception ( receive
**  all identifiers and put messages in receive queue 1 )
*/
BufDesc.MsgObjNum  = 15;
BufDesc.RxQueueNum = 1;
BufDesc.Identifier = 0;
BufDesc.Extended   = TRUE;

result = ioctl(fd, T903_DEFRXBUF, (char*)&BufDesc);


/*
**  ( 4.) Enter Bus On State
*/
result = ioctl(fd, T903_BUSON, NULL);
```

Now you should be able to send and receive CAN messages with appropriate calls to write() and read() functions.

4 Debugging and Diagnostic

If your installed IPAC port driver (e.g. tip903) doesn't find any devices although the IPAC is properly plugged on a carrier port, it's interesting to know what's going on in the system.

Usually all TEWS TECHNOLOGIES device driver announced significant event or errors via the device driver routine kkprintf(). To enable the debug output you must define the macro DEBUG in the device driver source files (e.g. carrier_class.c, carrier_tews_pci.c, tip903.c,...).

The debug output should appear on the console. If not please check the symbol KKPF_PORT in uparam.h. This symbol should be configured to a valid COM port (e.g. SKDB_COM1).

The following output appears at the LynxOS debug console if the carrier and IPAC driver starts:

```
TEWS TECHNOLOGIES - IPAC Carrier Class Driver version 1.2.0 (2005-04-05)
TEWS TECHNOLOGIES - VME Carrier version 1.2.0 (2005-04-05)
IPAC_CC  :  IPAC (Manuf-ID=B3, Model#=1C) recognized @ slot=0 carrier=<TEWS TECHNOLOGIES - VME
Carrier>
TIP903 Extended CAN driver version 2.1.0 (2009-03-02)
TIP903  :  Probe new TIP903 mounted on <TEWS TECHNOLOGIES - VME Carrier> at slot A
```

If driver installation was successful but you can't send or receive messages you should check the wiring and termination of the CANbus lines.

If all seems to be correct but you can't communicate over the CANbus please call the ioctl function T903_CANSTATUS direct after the write function returned to get extended error information (see also Intel 82527 Architectural Overview - 4.3 Status Register for detailed error description).

If you can't solve the problem by yourself, please contact TEWS TECHNOLOGIES with a detailed description of the error condition, your system configuration and the debug outputs.