SJA1000 CAN driver V1.8.1 for Linux 2.4.x

Cristiano Brudna

14th November 2003

1 Overview

This driver is written for a PCAN-Dongle[1] board that has an internal SJA1000 CAN controller. The driver supports both blocking and non-blocking modes of operation and allows to set the configuration of the SJA1000 such like baud rate and filtering properties (only single filter mode is supported).

The PCAN-Dongle is connected to the parallel interface of the PC and operates in the EPP mode, which must be set in the BIOS of the PC computer. The driver is compiled as a loadable kernel module and must be loaded before using any CAN application. The driver uses read interrupts generated by the SJA1000 to read new messages as soon as they arrive. It also uses write interrupts to receive an interrupt whenever a message has been sent and thus check for further messages still waiting to be sent. The driver has also a protection against reception of an overwhelming number of interrupts. Whenever a situation like this occurs, the SJA1000 interrupts are disabled preventing the system from crashing.

1.1 Instalation Instructions

Compile and install (as **root**) with: ’make install’.

The makefile will:

- create a node entry in ’dev/can’ and set the read and write permissions
- copy the driver module ’sja1000.o’ to ’/lib/modules/$(release)/misc’
- copy the include file ’sja1000.h’ to ’/usr/lib/can/can.h’

Compile the utils with: ’make utils’

1.1.1 Manual insertion

Load the module (as root): ’make insert’ or ’insmod sja1000.o’

The driver can be removed (as root) with ’make remove’ or ’rmmod sja1000’.

Two parameters can be given to the driver:

- irq: the IRQ of the parallel port (default: 7).
  Ex: ’insmod sja1000.o irq=5’.

[1] PCAN-Dongle/PS2: www.phytec.com

- port: the base I/O port of the parallel port (default: 0x278).
  Ex: 'insmod sja1000.o port=0x378'

1.1.2 Automatic insertion

To load it automatically at boot time add this line to '/etc/init.d/boot.local':
'insmod /lib/modules/$(shell uname -r)/misc/sja1000.o'
If the irq and/or port number are different, include the options according to the previous section.

1.1.3 Remarks

Information about the module can be obtained by the command **modinfo**.
Ex: 'modinfo sja1000.o'.
The driver also creates an entry in the proc file-system that can be read to get the current configuration and status. This can be done, for instance, with:
'less /proc/can'.

1.2 API

The driver provide 5 file operations. From version 1.7, `select()` is also supported.

1.2.1 CAN messages

CAN messages are defined as a struct with the following content:

```
typedef struct {
  CanId id;   // identifier (11 or 29 bits)
  int type;   // standard (0) or extended frame (1).
             //  Use the predefined values 'STAN-
DARD' and 'EXTENDED'.
  int rtr;   // remote transmission re-
quest (1 when true)
  int len;   // data length 0..8
  unsigned char d[8];  // data bytes
  struct timeval timestamp; // times-
tamp in the format
                            // times-
tamp.tv_sec (seconds)
                            // times-
tamp.tv_usec (microseconds)
                            // since Epoch (Jan-
uary 1. 1970).
} canmsg;
```

REMARK: Check the examples 'canmon.c' and 'cansend.c' to see the details.

1.2.2 open()

Open the device. The driver supports both blocking and non-blocking modes.

For Blocking mode use:

```
Ex:
    int can;
    can = open("/dev/can", O_RDWR);
```

The non-blocking mode must be specified in the open file operation with the flag *O_NONBLOCK* in the last parameter:

```
Ex: can = open("/dev/can", O_RDWR | O_NONBLOCK);
```

After `open()` the SJA1000 is set to 1Mbit/s with single filter mode. The acceptance mask is set to 0xffffffff, which means that all messages will be accepted.

WARNING: only **one** application is allowed to use the driver at a time. If the open operation is called by a second application an **-EBUSY** error code is returned.

1.2.3 close()

Close the device.

```
Ex: close(can);
```

1.2.4 read()

Read a message. Return **32** (size of the canmsg struct) when a message is available and **-EAGAIN** when there is no message.

```
Ex: read(can, &msg, sizeof(msg));
```

1.2.5 write()

Write a message. Return **32** (size of the canmsg struct) when the message is successfully stored in the driver's transmit buffer and **-EAGAIN** when the buffer is full.

```
Ex: write(can, &msg, sizeof(msg));
```

1.2.6 ioctl()

This command provides the following operations:

CAN_IOCSBAUD: set baud rate. Use the following constants to set it: **B1000** (1Mbit/s), **B500** (500kbit/s), **B250** (250kbit/s), **B125** (125kbit/s), **B20** (20kbit/s). The standard bit timing register values are:

{BTR0, BTR1}: baud rate
{0x00, 0x14}: 1 Mbit/s
{0x00, 0x1c}: 500 kbit/s
{0x01, 0x1c}: 250 kbit/s
{0x03, 0x1c}: 125 kbit/s
{0x18, 0x1c}: 20 kbit/s
REMARK*: See CAN_IOCSBTR*

```
Ex:
    unsigned long baud_rate = B1000;
    ioctl(can, CAN_IOCSBAUD, &baud_rate);
```

CAN_IOCSAMASK: set acceptance mask. Use a 32-bit value to set it.

```
Ex:
    long long amask=0xfffffffd;
    ioctl(can, CAN_IOCSAMASK, &amask);
```

CAN_IOCSACODE: set acceptance code. Use a 32-bit value to set it.

```
Ex:
    long long acode=0xfffffffe;
    ioctl(can, CAN_IOCSACODE, &acode);
```

CAN_IOCCRBUF: clear read buffer.

CAN_IOCCWBUF: clear write buffer.

CAN_IOCRREG: read a SJA1000 register. It can be any of the available registers. The most useful ones for applications are:

- ERROR_CODE_CAPTURE
- RX_ERROR_COUNTER
- TX_ERROR_COUNTER

```
Ex:
   int outcome;
   unsigned long reg;
   reg = RX_ERROR_COUNTER;
   outcome = ioctl(can, CAN_IOCRREG, &reg);
   printf("RX_ERROR_COUNTER = %dd\n", outcome);
```

CAN_IOCRTTS: read the timestamp of the last transmitted message. The timestamp is returned in a timeval structure.

CAN_IOCSACTIVE: set active mode

CAN_IOCSPASSIVE: set passive mode

CAN_IOCRAPS: get current active/passive status

CAN_IOCSBTR: set bit timing registers directly. The parameters are passed through the following struct:

```
typedef struct {
  unsigned char bt0;
  unsigned char bt1;
} canconfig;
Ex:
    canconfig bconfig;
    bconfig.bt0 = 0x0;
    bconfig.bt1 = 0x1c;
    ioctl(can, CAN_IOCSBTR, &bconfig);
```

1.2.7 select()

Example for checking for received messages:

```
ret = select(can+1, &readfds, NULL, NULL, &time-
out);
```

1.2.8 Remarks

Asynchronous notification is not supported yet.

2 Utils

All the utils allow to set the required bit rate. Use ’canmon -h’ to check the full options list.

canmon Receive CAN messages. It is possible to set filtering properties.

cansend Send standard/extended CAN messages.

busload Shows the bus load in messages per second.

cananalyser Classify messages in a per-CAN-ID manner. Shows CAN-ID, average rate (msg/s), average period (s), and total message count.

canping Ping tool for round-trip mesurement. Shows min, average, and max. round-trip time.

canreply Reply tool for round-trip measurement.

3 Remarks

To use this module with a different kind of board (ISA, PCI) it’s necessary to change the low level routines ’`can_w`’, to write to the board, and ’`can_r`’ to read from the board.