How to use the CAN with Linux

# Introduction

The Controller Area Network is a serial communications protocol which efficiently supports distributed real-time control with a high level of security. The SAMA5D3 CAN controller provides all the features required to implement the serial communication protocol.CAN defined by Robert Bosch GmbH, the CAN specification as referred to by ISO/11898A (2.0 Part A and 2.0 Part B) for high speeds and ISO/11519-2 for low speeds. The CAN Controller is able to handle all types of frames (Data, Remote, Error and Overload) and achieves a bitrate of 1 Mbit/sec

SocketCAN is the framework for CAN under Linux to replace plenty of vendor-specific CAN APIs. While the traditional CAN drivers for Linux are based on the model of character devices. Socket CAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The SocketCAN uses the model of network devices, which allows multiple applications to access one CAN device simultaneously. Also, a single application is able to access multiple CAN networks in parallel.

Since CAN is a networking interface and uses the socket layer concepts, many utilities have been developed in open source for utilizing CAN interface. These utils are very useful for debugging the driver.

* For testing CAN we commonly use the can-utils utility suite which includes cansend ,cangen and candump tools to send and receive packets via CAN interface.
* To configure the CAN interface netlink standard utilities are used and this requires iproute2 utilities.

This application note provides how to use the CAN controller in Linux on ATMEL SAMA5D3XEK.

## Iproute2

Iproute2 is a collection of utilities for controlling [TCP](http://en.wikipedia.org/wiki/Transmission_Control_Protocol) and [UDP](http://en.wikipedia.org/wiki/User_Datagram_Protocol) [IP](http://en.wikipedia.org/wiki/Internet_Protocol) networking and [traffic control](http://en.wikipedia.org/wiki/Network_traffic_control) in [Linux](http://en.wikipedia.org/wiki/Linux), in both [IPv4](http://en.wikipedia.org/wiki/IPv4) and [IPv6](http://en.wikipedia.org/wiki/IPv6) networks. It is currently maintained by [Stephen Hemminger](http://en.wikipedia.org/w/index.php?title=Stephen_Hemminger&action=edit&redlink=1). The original author, Alexey Kuznetsov, was responsible for the [QoS](http://en.wikipedia.org/wiki/Quality_of_service" \o "Quality of service) implementation in the [Linux kernel](http://en.wikipedia.org/wiki/Linux_(kernel)).

iproute2 is intended to replace an entire suite of standard Unix networking tools (often called “net-tools”) that were previously used for the tasks of configuring network interfaces, [routing](http://en.wikipedia.org/wiki/Routing) tables, and managing the [ARP](http://en.wikipedia.org/wiki/Address_Resolution_Protocol) table.

Homepage:

<http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2>

git tree

<git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git>

## CAN utilities

Can-utils is CAN user space utilities and tools for [SocketCAN](http://server/twiki/bin/edit/Shanghai/SocketCAN?topicparent=Shanghai.LinuxCAN;nowysiwyg=0).

Homepage:

<https://gitorious.org/linux-can/can-utils>

git tree:

git://gitorious.org/linux-can/can-utils.git

# 2. the CAN with SAMA5D3

## 2.1. SAMA5D3 CAN feature.

## 2.2. SocketCAN Software Architecture

SocketCAN software architecture shown in the figure below, is mainly divided into three parts, namely, user space, kernel space

Application

Socket Layer

Protocol Family (PF\_CAN)

CAN Networking Driver

User Space

Kernel Space

In order to bring CAN networking to the Linux kernel CAN support has been added to the existing networking subsystem. This primarily consists of two parts:

1. A new protocol family PF\_CAN including a CAN\_RAW protocol,
2. The drivers for various CAN networking devices.

Socket interface provides a socket interface to user space applications and which builds upon the Linux network layer. CAN networking driver for CAN controller hardware registers itself with the Linux network layer as a network device. So that CAN frames from the controller can be passed up to the network layer and on to the CAN protocol family module and vice-versa.

CAN networking drivers implement the same standardized networking driver model as Ethernet drivers, So writing a CAN networking device driver is much easier than writing a CAN character device driver. Similar to other known network device drivers you mainly have to deal with:

- TX: Put the CAN frame from the socket buffer to the CAN controller.

- RX: Put the CAN frame from the CAN controller to the socket buffer.

## 2.3. Linux Driver Configuration

SAMA5D3 CAN device driver in Linux is provided as a networking driver that confirms to the socketCAN interface. The driver is currently build-into the kernel with the right configuration items enabled.

It is available as a configuration item in the Linux kernel configured as follows.

[\*] Networking support -🡪

<\*> CAN bus subsystem support -🡪

<\*> Raw CAN Protocol (raw access with CAN-ID filtering)

<\*> Broadcast Manager CAN Protocol (with content filtering)

CAN Device Drivers -🡪

<\*> Virtual Local CAN Interface (vcan)

<\*> Platform CAN drivers with Netlink support

[\*] CAN bit-timing calculation

<\*> Atmel AT91 onchip CAN controller

Note that if the kernel configuration for “CAN bit-timing calculation” is not enabled then each of the parameters: tq, PROP\_SEG etc need to be set individually. When bit-timing calculation is enabled in the kernel configuration, the CAN bit-timing parameters will be calculated if the bit-rate is specified with the argument “bitrate”.

## 2.4. Buildroot Configuration

The iproute2 and can-utils utilities package has been supported in the latest version of buildroot(we use the version 2013.02). They can be compiled and installed on the target by means of the Buildroot menuconfig.

Package Selection for the target -🡪

Networking applications -🡪

[\*] can-utils

[\*] iproute2

After build kernel and rootfs, program them into the board, then power up.

To Verify CAN support configuration, You can use the command to look for the following in the Linux boot output

# dmesg | grep can

vcan: Virtual CAN interface driver

at91\_can f000c000.can: device registered (reg\_base=e0d28000, irq=20)

at91\_can f8010000.can: device registered (reg\_base=e0d2a000, irq=28)

qt1070 1-001b: can not read register, returned -121

can: controller area network core (rev 20120528 abi 9)

can: raw protocol (rev 20120528)

can: broadcast manager protocol (rev 20120528 t)

Now, the at91\_can device works. Let’s use it

# 3. Use CAN

## 3.1. Prepare Hardware

There are two CAN interfaces on the SAMA5D3X-EK board. The CAN interface is available on RJ11 connectors. The CAN pins details are as following table.

|  |  |
| --- | --- |
| Pin | Signal |
| 4 | CANL |
| 5 | CANH |
| 1,2 | VCC |
| 3,6 | GND |

To connect the two CAN interface, the following connections need to made. CAN0 connector pins 4, 5 needs to be connected to pins 4,5 of CAN1 connector respectively. Since CAN0 and CAN1 is on the same boards, the VCC and GND is no need to connect.

5

4

4

5

CANL

CANH

CAN0

CAN1

SAMA5D3X-EK

## 3.2. Configure CAN

* Examine the CAN network interface using command.

# /sbin/ip -details link show | grep can

2: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN mode DEFAULT qlen 10

link/can

can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0

at91\_can: tseg1 4..16 tseg2 2..8 sjw 1..4 brp 2..128 brp-inc 1

3: can1: <NOARP,ECHO> mtu 16 qdisc noop state DOWN mode DEFAULT qlen 10

link/can

can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0

at91\_can: tseg1 4..16 tseg2 2..8 sjw 1..4 brp 2..128 brp-inc 1

From above information, there are two CAN network interfaces available, can0 and can1.

* Set the bitrate before all operations

Set the can0 and can1 bit rate to 125Kbits/sec

# /sbin/ip link set can0 type can bitrate 125000

# /sbin/ip link set can1 type can bitrate 125000

Once the driver installed and the bitrate is set, the CAN interface has to be started like a standard net interface

* Bring up can0 and can1 using the command:

# /sbin/ip link set can0 up

at91\_can f000c000.can: can0: writing AT91\_BR: 0x00200561

# /sbin/ip link set can1 up

at91\_can f8010000.can: can1: writing AT91\_BR: 0x00200561

## Transfer and Receive packets

To test the CAN interface, transfer and receive simple packets by using cansend and candump utility.

* **candump**: dump traffic on a CAN network
* **cansend**: simple command line tool to send CAN-frames via CAN\_RAW sockets

To receive the CAN data frame package, run candump on the can1 interface.

To Transfer a CAN data frame message, with a can\_id arbitration field value of 0x123 and a data field value 0x11,0x22,0x33, 0x44, 0x55, 0x66, 0x77, 0x88, run cansend on the can0 interface.

# candump can1 &

# cansend can0 123#11.22.33.44.55.66.77.88

can1 123 [8] 11 22 33 44 55 66 77 88

The cansend utility usage:

# cansend ---help

Usage: cansend <device> <can\_frame>.

where the device is the CAN network interface name, typically can0, and a CAN frame is in the format:

<can\_id>#{R|data}

with the can\_id having 3 (SFF) or 8 (EFF) hex chars. and data in the format of zero to eight 8-bit hex-values that can optionally be separated by a period ('.') or use R for remote transmission request.

To retrieve the socketCAN version using command.

# cat /proc/net/can/version

rev 20120528 abi 9

To retrieve the socketCAN statistics using command.

# cat /proc/net/can/stats

## 3.4. Application Example

Give a simple application example using SocketCAN.

### SocketCAN initialization

int sock; /\* can raw socket \*/

struct sockaddr\_can addr;

struct ifreq ifr;

/\* open socket \*/

if ((sock = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW)) < 0) {

perror("socket");

return 1;

}

addr.can\_family = AF\_CAN;

strcpy(ifr.ifr\_name, "can0");

if (ioctl(sock, SIOCGIFINDEX, &ifr) < 0) {

perror("SIOCGIFINDEX");

return 1;

}

addr.can\_ifindex = ifr.ifr\_ifindex;

if (bind(sock, (struct sockaddr \*)&addr, sizeof(addr)) < 0) {

perror("bind");

return 1;

}

### Send CAN message

frame.can\_id = 0x123;

frame.can\_dlc = 1;

frame.data[0] = 0xAB;

/\* send frame \*/

nbytes = write(sock, &frame, sizeof(frame));

### Receive CAN message

/\* read frame \*/

nbytes = read(sock, &frame, sizeof(struct can\_frame));

if (nbytes < 0) {

perror("can raw socket read");

return 1;

}

printf("ID = 0x%x DLC = %d, data[0] = 0x%x\n",

frame.can\_id,

frame.can\_dlc,

frame.data[0]);

### RAW Socket with can\_filters

The reception of CAN frames using CAN\_RAW sockets can be controlled by defining 0 .. n filters with the CAN\_RAW\_FILTER socket option.

SocketCAN does not support hardware filtering of incoming CAN frames. Currently all CAN frames are received and passed to the CAN networking layer core, which processes the application specific filter lists. Activation of hardware filters would lead to an overall reduction of the received CAN traffic, but are a global setting. In a multi-user, multi-application scenario hardware filters are not feasible.The high efficient filter sets inside the PF\_CAN core allow to set different multiple filters for each socket separately.

struct can\_filter rfilter[2];

rfilter[0].can\_id = 0x123;

rfilter[0].can\_mask = CAN\_SFF\_MASK;

rfilter[1].can\_id = 0x200;

rfilter[1].can\_mask = 0x700;

setsockopt(sock, SOL\_CAN\_RAW, CAN\_RAW\_FILTER, &rfilter, sizeof(rfilter));

# Conclusion