Category:CAN

Contents

Stable: 17.06.2020 - 15:26 / Revision: 16.01.2020 - 08:40

Category:CAN

Stable: 17:06:2020 - 10:20 | Revision: 16:01:2020 - 08:40

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to the Linux® **CAN** software framework.

It is recommended to first read the CAN overview article.

Linux® is a registered trademark of Linus Torvalds.

Controller Area Network (robust bus mainly used for automotive applications)

Pages in category "CAN"

The following 4 pages are in this category, out of 4 total.

- CAN overview
- FDCAN device tree configuration
- How to send or receive CAN data
- How to set up a SocketCAN interface

Stable: 01.12.2020 - 10:52 / Revision: 10.06.2020 - 14:46

A quality version of this page, approved on *1 December 2020*, was based off this revision.

This article gives information about the Linux® Controller Area Network (CAN) framework. It explains how to activate the CAN interface and, based on examples, how to use it.

Contents

1 Framework purpose

The **Controller Area Network** (CAN) is a multi-master serial bus standard connecting at least two nodes. It is a message-based protocol originally designed for in-vehicle communication and which main benefits are a significant reduction of wiring and the prevention of message collision.

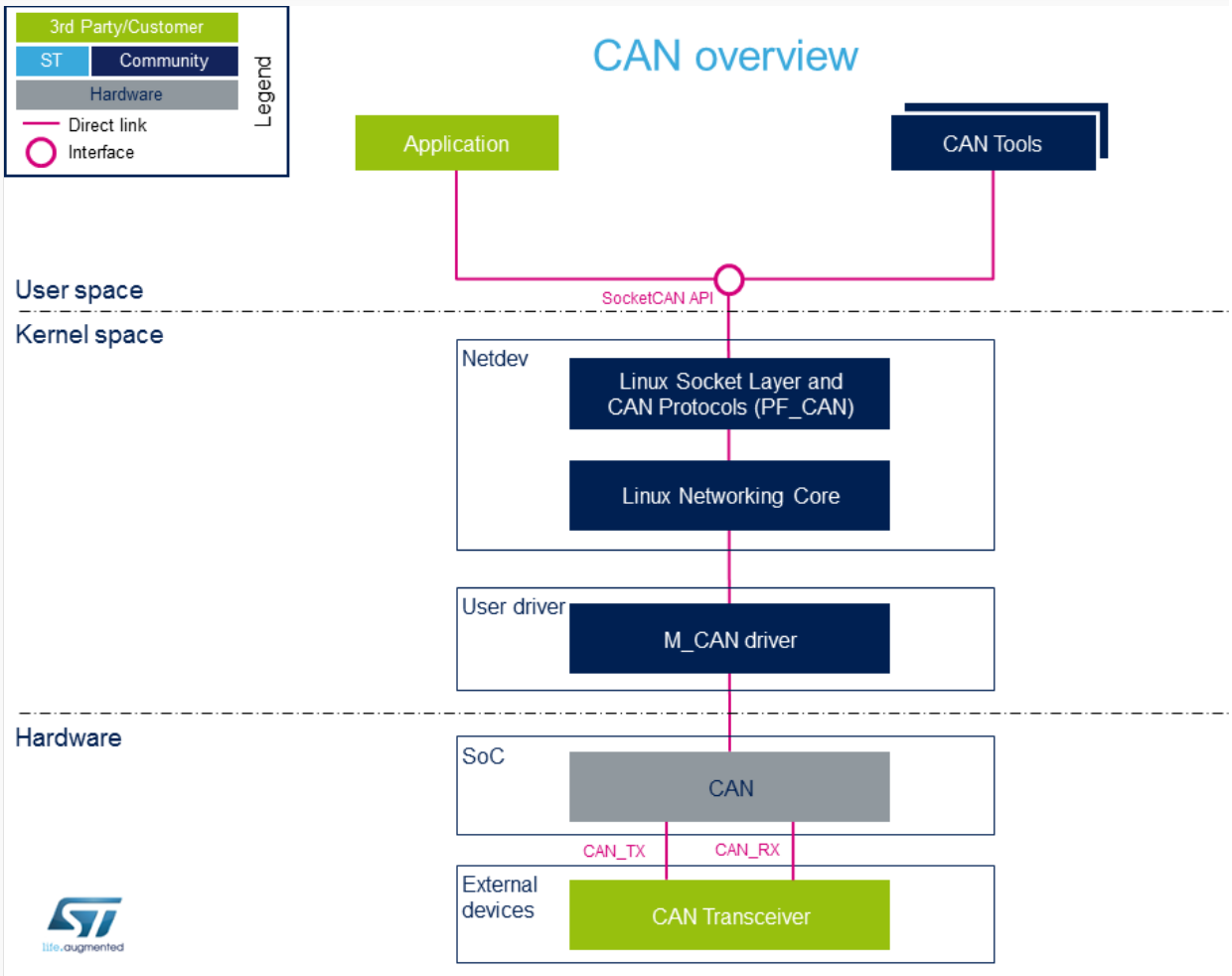
For better real-time performance, CAN with Flexible Data-Rate (CAN FD)[1] is used as an extension to the classic CAN protocol [2]. It allows data rates higher than 1 MBit/s and payloads longer than 8 bytes per frame (up to 64 data bytes).

SocketCAN [3] is a uniform CAN Framework for the Linux kernel. It implements a new protocol family called **PF_CAN**[4] and allows applications to receive and transmit CAN messages via Socket APIs with CAN specific socket options.

You can find many applications of CAN in the automotive industry. In vehicles, it allows electronic control units and devices to communicate with each other in applications without a host computer. For example, a high speed CAN bus is dedicated to security devices such as emergency brake system or airbags. Another low speed CAN bus is dedicated to comfort devices such as interiror lighting or seat control.

This article describes the main components and APIs of the CAN Framework and gives examples of CAN usage.

2 System overview



CAN Overview

2.1 Component description

From user space to hardware

- **Application** (User space)

Application to read/write data on the SocketCAN interface for communication with external devices connected on the CAN network (such as can-utils).

- **CAN tools** (User space)

Set of utilities for configuring and enabling SocketCAN interface (such as iproute2).

- **SocketCAN** (Kernel space)

Socket interface with specific CAN options which builds upon the Linux network layer.

- **Linux Socket Layer and CAN Protocols (PF_CAN)** (Kernel space)

The protocol family, PF_CAN[4], provides an API for transport protocol modules to register and the structures to enable different CAN protocols on the bus.

- **Linux Networking Core** (Kernel space)

Kernel network layer that adapts the message to the transport protocol in use. The network subsystem of the Linux kernel is designed to be completely protocol-independent.

- **M_CAN Driver** (Kernel space)

Driver implemented as a network interface for Bosch M_CAN controller[5].

- **CAN** (Hardware)

This is the CAN Core IP.

- **CAN Transceiver** (Hardware)

Interface between the CAN protocol controller and the physical wires of the CAN bus lines.

2.2 API description

The SocketCAN interface API description can be found in kernel documentation [3].

3 Configuration

3.1 Kernel configuration

Activate the CAN driver in kernel configuration with Linux Menuconfig tool.

For compiling M_CAN driver, select "Bosch M_CAN devices":

```
 [*] Networking support  ---
>
        <*>   CAN bus subsystem support  --->
                CAN Device Drivers  --->
                        <*>   Bosch M_CAN support
                        <*>       Bosch M_CAN support for io-mapped devices
```

M_CAN driver is activated by default in ST deliveries.

3.2 Device tree configuration

CAN generic DT bindings:

- The M_CAN device tree bindings[6] describe all the required and optional properties.

Detailed DT configuration for STM32 internal peripherals:

- FDCAN device tree configuration

4 How to use the framework

The CAN device must be configured via netlink interface. The following articles give user space examples of how to set up a SockeCAN interface (and configure settings like bit-timing parameters) and how to send/receive data on the CAN bus.

4.1 How to set up a SocketCAN interface

How to set up a SocketCAN interface

4.2 How to send/receive CAN data

How to send or receive CAN data

5 How to trace and debug the framework

5.1 How to trace

CAN Framework, specifically M_CAN driver, print out info and error messages. You can display them with dmesg command:

```
Board $> dmesg | grep m_can
[    1.327824] m_can 4400e000.can: m_can device registered (irq=30, version=32)
[   25.560759] m_can 4400e000.can can0: bitrate error 0.3%
[   25.564630] m_can 4400e000.can can0: bitrate error 1.6%
```

5.2 How to monitor CAN bus

You can use the CAN FD adapter **PCAN-USB Pro FD**[7] to connect a computer to the CAN network via USB. The PCAN-View software provided with the tool is a monitoring program that allows to supervise the data flow on the CAN network and to detect frame errors.

6 Source code location

The source files are located inside the Linux kernel.

- **PF_CAN**: af_can.c[4]
- **M_CAN driver**: m_can.c[5]

7 To go further

CAN bit timing calculation plays an important role in ensuring performance of CAN network. To avoid transmission errors, the bit timing must be configured properly.

For more information about CAN bit timing:

- *Computation of CAN Bit Timing Parameters Simplified*[8], from the CAN in Automation group (CiA)
- *The Configuration of the CAN Bit Timing*[9], from Bosch documentation

8 References

- CAN FD - The basic idea, from the CAN in Automation group (CiA)
- CAN protocol implementations, from the CAN in Automation group (CiA)
- 3.03.1 Kernel SocketCAN documentation , Linux Foundation
- 4.04.14.2 net/can/af_can.c , Protocol family CAN core module
- 5.05.1 drivers/net/can/m_can/ , Driver for Bosch M_CAN controller
- Documentation/devicetree/bindings/net/can/m_can.txt M_CAN device tree bindings
- PCAN-USB Pro FD description, by PEAK System
- Computation of CAN Bit Timing Parameters Simplified, from the CAN in Automation group (CiA)
- The Configuration of the CAN Bit Timing, from Bosch documentation

Controller Area Network (robust bus mainly used for automotive applications)

Application programming interface

Device Tree

Stable: 01.12.2020 - 10:53 / Revision: 10.06.2020 - 14:35

A quality version of this page, approved on *1 December 2020*, was based off this revision.

Contents

1 Article purpose

This article explains how to configure the FDCAN when it is assigned to the Linux®OS. In that case, it is controlled by the CAN framework for Bosch M_CAN controller.

The configuration is performed using the device tree mechanism that provides a hardware description of the FDCAN peripheral, used by the M_CAN Linux driver and by the NET/CAN framework.

If the peripheral is assigned to another execution context, refer to How to assign an internal peripheral to a runtime context article for guidelines on peripheral assignment and configuration.

2 DT bindings documentation

M_CAN device tree bindings[1] describe all the required and optional properties.

3 DT configuration

This hardware description is a combination of the **STM32 microprocessor** device tree files (*.dtsi* extension) and **board** device tree files (*.dts* extension). See the Device tree for an explanation of the device tree file split.

STM32CubeMX can be used to generate the board device tree. Refer to How to configure the DT using STM32CubeMX for more details.

3.1 DT configuration (STM32 level)

All M_CAN nodes are described in stm32mp153.dtsi [2] file with disabled status and required properties such as:

- Physical base address and size of the device register map
- Message RAM address and size (CAN SRAM)
- Host clock and CAN clock
- Message RAM configuration

This is a set of properties that may not vary for a given STM32 device.

```
 m_can1: can@4400e000 {
         compatible = "bosch,m_can";
         reg = <0x4400e000 0x400>, <0x44011000 0x1400>;    /* FDCAN1 uses only the first
half of the dedicated CAN_SRAM */
         reg-names = "m_can", "message_ram";
         interrupts = <GIC_SPI 19 IRQ_TYPE_LEVEL_HIGH>,
                      <GIC_SPI 21 IRQ_TYPE_LEVEL_HIGH>;
         interrupt-names = "int0", "int1";
         clocks = <&rcc CK_HSE>, <&rcc FDCAN_K>;
         clock-names = "hclk", "cclk";
         bosch,mram-cfg = <0x0 0 0 32 0 0 2 2>;
         status = "disabled";
 };

 m_can2: can@4400f000 {
         compatible = "bosch,m_can";
         reg = <0x4400f000 0x400>, <0x44011000 0x2800>;    /* The 10 Kbytes of the CAN_SRA
M are mapped */
         reg-names = "m_can", "message_ram";
         interrupts = <GIC_SPI 20 IRQ_TYPE_LEVEL_HIGH>,
                      <GIC_SPI 22 IRQ_TYPE_LEVEL_HIGH>;
         interrupt-names = "int0", "int1";
         clocks = <&rcc CK_HSE>, <&rcc FDCAN_K>;
         clock-names = "hclk", "cclk";
         bosch,mram-cfg = <0x1400 0 0 32 0 0 2 2>;         /* Set mram-cfg offset to
write FDCAN2 data on the second half of the dedicated CAN_SRAM */
         status = "disabled";
 };
```

The required and optional properties are fully described in the bindings files.

Warning

This device tree part is related to STM32 microprocessors. It must be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

Part of the device tree is used to describe the FDCAN hardware used on a given board. The DT node (**"m_can"**) must be filled in:

- Enable the CAN block by setting **status = "okay".**
- Configure the pins in use via pinctrl, through **pinctrl-0** (default pins), **pinctrl-1** (sleep pins) and **pinctrl-names**.

3.3 DT configuration examples

The example below shows how to configure and enable FDCAN1 instance at board level:

```
 &m_can1 {
         pinctrl-names = "default", "sleep";          /* configure pinctrl modes for
m_can1 */
         pinctrl-0 = <&m_can1_pins_a>;                /* configure m_can1_pins_a as
default pinctrl configuration for m_can1 */
         pinctrl-1 = <&m_can1_sleep_pins_a>;          /* configure m_can1_sleep_pins_a as
sleep pinctrl configuration for m_can1 */
         status = "okay";                             /* enable m_can1 */
 };
```

4 How to configure the DT using STM32CubeMX

The STM32CubeMX tool can be used to configure the STM32MPU device and get the corresponding platform configuration device tree files.
The STM32CubeMX may not support all the properties described in the above DT bindings documentation paragraph. If so, the tool inserts **user sections** in the generated device tree. These sections can then be edited to add some properties and they are preserved from one generation to another. Refer to STM32CubeMX user manual for further information.

5 References

Please refer to the following links for additional information:

- Documentation/devicetree/bindings/net/can/m_can.txt M_CAN device tree bindings
- arch/arm/boot/dts/stm32mp153.dtsi , STM32MP153 device tree file

Operating System

Controller Area Network (robust bus mainly used for automotive applications)

Device Tree

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Generic Interrupt Controller

Serial Peripheral Interface

High Speed External oscillator (STM32 clock source)

Stable: 03.02.2020 - 07:57 / Revision: 03.02.2020 - 07:49

A quality version of this page, approved on *3 February 2020*, was based off this revision.

Contents

1 Purpose

This article describes how to send/receive data on a SocketCAN interface using the **can-utils**[1] package.

The **can-utils** contains some userspace utilities for Linux® SocketCAN subsystem. It is integrated in the SDK for the STM32 microprocessor Series. Only **cansend** and **candump** are used in this example, but many other tools are available in the package.

2 Prerequisite

At least two nodes are required to communicate on a CAN network. To implement such configuration, connect two boards on the same CAN bus. Then, send data from one node, and receive data on the other node.

Prior to starting communications, the SocketCAN interface has to be configured and enabled on each board. See How to set up a SocketCAN interface.

3 Sending data on the CAN bus

To send a single frame, use the **cansend** utility:

```
Board $> cansend can0 123#1122334455667788
```

To print help on **cansend** utility:

```
Board $> cansend -h
Usage: cansend <device> <can_frame>
```

If you get frame error, try:

```
    <can_id>#{R|data}          for CAN 2.0 frames
    <can_id>##<flags>{data}    for CAN FD frames

<can_id> can have 3 (SFF) or 8 (EFF) hex chars
{data} has 0..8 (0..64 CAN FD) ASCII hex-values (optionally separated by '.')
<flags> a single ASCII Hex value (0 .. F) which defines canfd_frame.flags

e.g. 5A1#11.2233.44556677.88 / 123#DEADBEEF / 5AA# / 123##1 / 213##311
     1F334455#1122334455667788 / 123#R for remote transmission request.
```

4 Receiving data on the CAN bus

To display in real-time the list of messages received on the bus, use the candump utility:

```
Board $> candump can0
can0  123   [8] 11 22 33 44 55 66 77 88
```

To print help on **candump** utility:

```
Board $> candump -h

Usage: candump [options] <CAN interface>+
   (use CTRL-C to terminate candump)

Options: -t <type>   (timestamp: (a)bsolute/(d)elta/(z)ero/(A)bsolute w date)
         -c          (increment color mode level)
         -i          (binary output - may exceed 80 chars/line)
         -a          (enable additional ASCII output)
         -S          (swap byte order in printed CAN data[] - marked with '`' )
         -s <level>  (silent mode - 0: off (default) 1: animation 2: silent)
         -b <can>    (bridge mode - send received frames to <can>)
         -B <can>    (bridge mode - like '-b' with disabled loopback)
         -u <usecs>  (delay bridge forwarding by <usecs> microseconds)
         -l          (log CAN-frames into file. Sets '-s 2' by default)
         -L          (use log file format on stdout)
         -n <count>  (terminate after receiption of <count> CAN frames)
         -r <size>   (set socket receive buffer to <size>)
         -D          (Don't exit if a "detected" can device goes down.
         -d          (monitor dropped CAN frames)
         -e          (dump CAN error frames in human-readable format)
         -x          (print extra message infos, rx/tx brs esi)
         -T <msecs>  (terminate after <msecs> without any reception)

Up to 16 CAN interfaces with optional filter sets can be specified on the commandline in
the form: <ifname>[,filter]*

Comma separated filters can be specified for each given CAN interface:
 <can_id>:<can_mask> (matches when <received_can_id> & mask == can_id & mask)
 <can_id>~<can_mask> (matches when <received_can_id> & mask != can_id & mask)
 #<error_mask>       (set error frame filter, see include/linux/can/error.h)
 [j|J]               (join the given CAN filters - logical AND semantic)

CAN IDs, masks and data content are given and expected in hexadecimal values.
When can_id and can_mask are both 8 digits, they are assumed to be 29 bit EFF.
Without any given filter all data frames are received ('0:0' default filter).

Use interface name 'any' to receive from all CAN interfaces.

Examples:
candump -c -c -ta can0,123:7FF,400:700,#000000FF can2,400~7F0 can3 can8
candump -l any,0~0,#FFFFFFFF    (log only error frames but no(!) data frames)
candump -l any,0:0,#FFFFFFFF    (log error frames and also all data frames)
candump vcan2,92345678:DFFFFFFF (match only for extended CAN ID 12345678)
candump vcan2,123:7FF (matches CAN ID 123 - including EFF and RTR frames)
candump vcan2,123:C00007FF (matches CAN ID 123 - only SFF and non-RTR frames)
```

5 Hardware self-test

In internal Loopback test mode, the FDCAN handles the messages it transmitted as received messages. This option is used for harware self-test (no need to connect an external CAN node on the CAN bus).

To configure and enable the SocketCAN in Loopback mode, proceed as follows:

```
Board $> ip link set can0 up type can bitrate 1000000 dbitrate 2000000 fd on loopback on
[ 78.700698] m_can 4400e000.can can0: bitrate error 0.3%
[ 78.704568] m_can 4400e000.can can0: bitrate error 1.6%
[ 78.710140] IPv6: ADDRCONF(NETDEV_CHANGE): can0: link becomes ready
```

To send and receive messages on the same interface, proceed as follows:

```
Board $> candump can0 -L &
[1] 475
Board $> cansend can0 300#AC.AB.AD.AE.75.49.AD.D1
(1539944874.949723) can0 300#ACABADAE7549ADD1
(1539944874.949683) can0 300#ACABADAE7549ADD1
```

6 References

- CAN-UTILS information, README on github.com

Linux® is a registered trademark of Linus Torvalds.

Software development kit (A programming package that enables a programmer to develop applications for a specific platform.)

Controller Area Network (robust bus mainly used for automotive applications)

Stable: 07.01.2021 - 13:37 / Revision: 20.11.2020 - 15:05

A quality version of this page, approved on *7 January 2021*, was based off this revision.

Contents

1 Purpose

This article describes how to set up a SocketCAN interface using the **iproute2**[1] suite of tools.

iproute2 is a set of utilities for Linux® networking, integrated in the SDK for the STM32 microprocessors.

2 Configuring the SocketCAN interface

The available CAN devices are listed in /sys/class/net/:

```
Board $> ls /sys/class/net
can0  eth0                /* can0 interface is available but not necessarily active */
```

You can also display all the available network interfaces to find out the available CAN devices:

```
Board $> ifconfig -a
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:30

eth0      Link encap:Ethernet  HWaddr 00:80:E1:42:45:EC
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:54 Base address:0x6000
```

Configure the available SocketCAN interface using the ip link command line as follow:

```
Board $> ip link set can0 type can bitrate 1000000 dbitrate 2000000 fd on
[ 78.700698] m_can 4400e000.can can0: bitrate error 0.3%
[ 78.704568] m_can 4400e000.can can0: bitrate error 1.6%
```

To list CAN user-configurable options, use the following command line:

```
Board $> ip link set can0 type can help

Usage: ip link set DEVICE type can
        [ bitrate BITRATE [ sample-point SAMPLE-POINT] ] |
        [ tq TQ prop-seg PROP_SEG phase-seg1 PHASE-SEG1
          phase-seg2 PHASE-SEG2 [ sjw SJW ] ]

        [ dbitrate BITRATE [ dsample-point SAMPLE-POINT] ] |
        [ dtq TQ dprop-seg PROP_SEG dphase-seg1 PHASE-SEG1
          dphase-seg2 PHASE-SEG2 [ dsjw SJW ] ]

        [ loopback { on | off } ]
        [ listen-only { on | off } ]
        [ triple-sampling { on | off } ]
        [ one-shot { on | off } ]
        [ berr-reporting { on | off } ]
        [ fd { on | off } ]
        [ fd-non-iso { on | off } ]
        [ presume-ack { on | off } ]
```

```
        [ restart-ms TIME-MS ]
        [ restart ]

        Where: BITRATE  := { 1..1000000 }
                  SAMPLE-POINT  := { 0.000..0.999 }
                  TQ            := { NUMBER }
                  PROP-SEG      := { 1..8 }
                  PHASE-SEG1    := { 1..8 }
                  PHASE-SEG2    := { 1..8 }
                  SJW           := { 1..4 }
                  RESTART-MS    := { 0 | NUMBER }
```

Warning

If the parameters **sjw** and **dsjw** are not specified with the **ip link** command, the Linux kernel will assign them with the default value 1. This default value is often too small and can cause receive errors if the bitrate of the transmitter is slightly different from the bitrate of the receiver.

3 Printing SocketCAN information

To get a detailed status of the SocketCAN link, use the following command line:

```
Board $> ip -details link show can0
2: can0: <NOARP,ECHO> mtu 72 qdisc pfifo_fast state DOWN mode DEFAULT group default qlen
10
    link/can  promiscuity 0
    can <FD> state STOPPED (berr-counter tx 0 rx 0) restart-ms 0
          bitrate 996078 sample-point 0.745
          tq 19 prop-seg 18 phase-seg1 19 phase-seg2 13 sjw 1
          m_can: tseg1 2..256 tseg2 1..128 sjw 1..128 brp 1..512 brp-inc 1
          dbitrate 2032000 dsample-point 0.720
          dtq 19 dprop-seg 8 dphase-seg1 9 dphase-seg2 7 dsjw 1
          m_can: dtseg1 1..32 dtseg2 1..16 dsjw 1..16 dbrp 1..32 dbrp-inc 1
          clock 50800000numtxqueues 1 numrxqueues 1 gso_max_size 65536 gso_max_segs 65535
```

4 Enabling/disabling the SocketCAN interface

Then enable the connection by bringing the SocketCAN interface up:

```
Board $> ip link set can0 up
```

You can check that the interface is up by printing the netlink status:

```
Board $>  ip -details link show can0
2: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 72 qdisc pfifo_fast state UNKNOWN mode DEFAULT
group default qlen 10
    link/can  promiscuity 0
    can <FD> state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0
          bitrate 996078 sample-point 0.745
...
```

You can disable the connection by bringing the SocketCAN interface down. This command is useful when you need to reconfigure the SocketCAN interface:

```
Board $> ip link set can0 down
```

5 Loopback test mode

It is possible to configure the SocketCAN in internal Loopback test mode. In that case, the FDCAN treats its own transmitted messages as received messages. This mode can be used for hardware self-test:

```
Board $> ip link set can0 type can bitrate 1000000 dbitrate 2000000 fd on loopback on
```

You can check that loopback option is on by printing the netlink status:

```
Board $> ip  -details link show can0
2: can0: <NOARP,ECHO> mtu 72 qdisc pfifo_fast state DOWN mode DEFAULT group default qlen
10
    link/can  promiscuity 0
    can <LOOPBACK,FD> state STOPPED (berr-counter tx 0 rx 0) restart-ms 0
          bitrate 996078 sample-point 0.745
...
```

6 Example of SocketCAN interface setup

You can configure and enable the SocketCAN interface in the same command line:

```
Board $> ip link set can0 up type can bitrate 1000000 dbitrate 2000000 fd on
[ 78.700698] m_can 4400e000.can can0: bitrate error 0.3%
[ 78.704568] m_can 4400e000.can can0: bitrate error 1.6%
[ 78.710140] IPv6: ADDRCONF(NETDEV_CHANGE): can0: link becomes ready
```

7 References

- IPROUTE2 information, by The Linux Foundation

Linux® is a registered trademark of Linus Torvalds.

Software development kit (A programming package that enables a programmer to develop applications for a specific platform.)

Controller Area Network (robust bus mainly used for automotive applications)

Receive

Transmit

uniprocessor