AM335X DCAN Driver Guide

AM335X DCAN Linux Driver Guide

Linux PSP

Introduction

The Controller Area Network is a serial communications protocol which efficiently supports distributed real-time control with a high level of security. The DCAN module supports bitrates up to 1 Mbit/s and is compliant to the CAN 2.0B protocol specification. The core IP within DCAN is provided by Bosch.

This wiki page provides usage information of DCAN Linux driver on AM335x EVM.

Acronyms & definitions

DCAN Driver: Acronyms

Acronym	Definition
CAN	Controller Area Network
BTL	Bit timing logic
DLC	Data Length Code
MO	Message Object
LEC	Last Error Code
FSM	Finite State Machine
CRC	Cyclic Redundancy Check

Setup Details

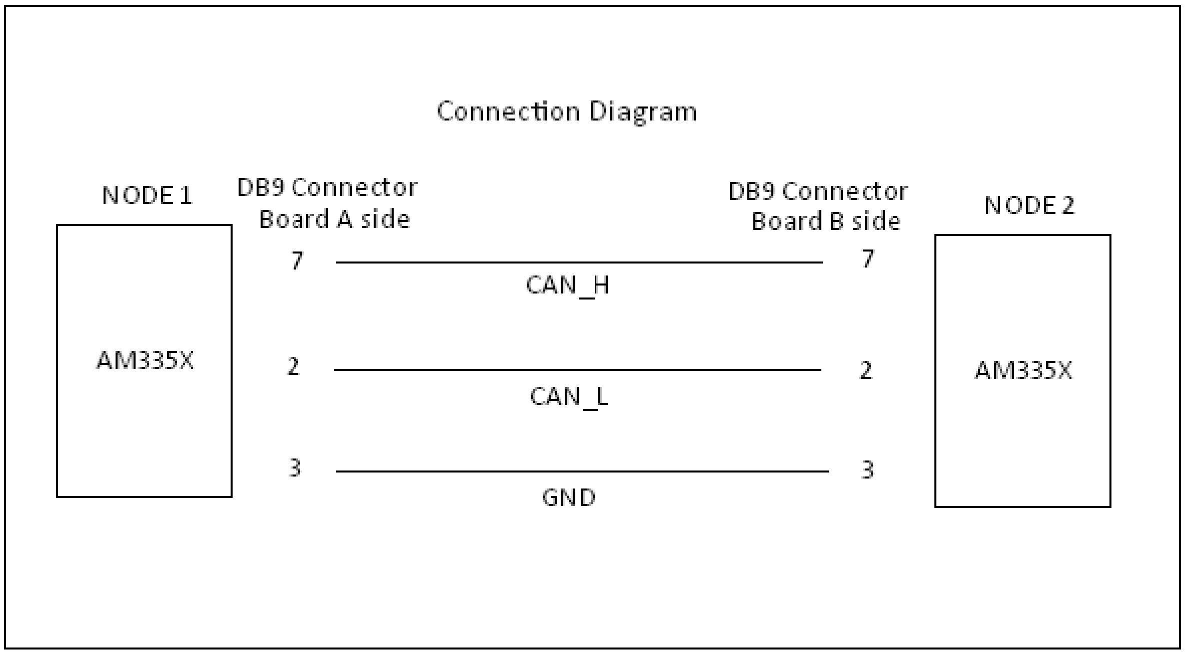
Connection details

am335x

- DCAN interface is available on DB9 connectors on the daughter board. The DCAN pin details are as follows:

1. Pin 7 - CAN-H
2. Pin 2 - CAN-L
3. Pin 3 - Ground

  - DB9F-DB9F straight cable for board-board connection is required and details can be found from digikey [1] website & corresponding datasheet [2]
  - To connect two EVM's then the following connections needs to be made. Board A connecter pins 7,2,3 needs to be connected to pins 7,2,3 of Board B connector

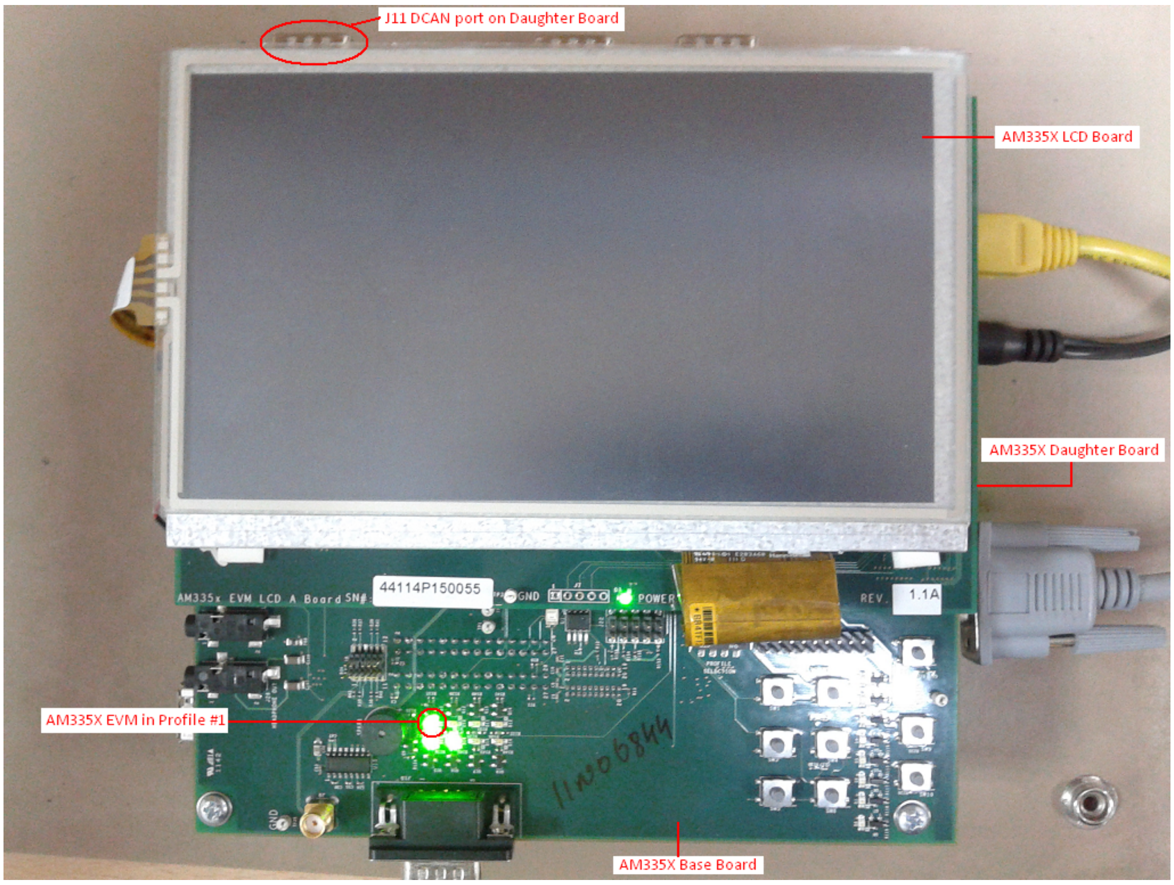


am335x-am335x connection diagram

Port details

am335x

- DCAN is available in Profile #1 of AM335X EVM with General purpose daughter card, go through EVM hardware user guide [3] to know how to set different profiles. Transceiver port number on the daughter card is J11 and below figure is for reference



AM335X DCAN port details

Driver Usage

Quick Steps

Utilities required for DCAN driver usage are ip, canconfig, cansend, candump and cansequence. Details of these utilities are elaborated in CAN Utilities section

Steps for transmitting/receiving CAN packets

- Set bit-timing

Set the bit-rate to 50Kbits/sec with triple sampling using the following command

```
$ ip link set can0 type can bitrate 50000 triple-sampling on
```

or

```
$ canconfig can0 bitrate 50000 ctrlmode triple-sampling on
```

- Device bring up

Bring up the device using the command:

```
$ ip link set can0 up
```

or

```
$ canconfig can0 start
```

- Transfer packets

Packet transmission can be achieve by using cansend and cansequence utilities.

a. Transmit 8 bytes with standard packet id number as 0x10

```
$ cansend can0 -i 0x10 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88
```

b. Transmit 8 bytes with extended packet id number as 0x800

```
$ cansend can0 -i 0x800 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 -e
```

c. Transmit 20 packets of 8 bytes each with same extended packet id number as 0xFFFFF

```
$ cansend can0 -i 0xFFFFF 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 -e --loop=20
```

d. Transmit a sequence of numbers from 0x00-0xFF, till the buffer availability

```
$ cansequence can0
```

e. Transmit a sequence of numbers from 0x00-0xFF and roll-back in a continuous loop

```
$ cansequence can0 -p
```

- Receive packets

Packet reception can be achieve by using candump utility

```
$ candump can0
```

Advanced Usage

Network up/down

- Bring up the device

```
$ ip link set can0 up
```

or

```
$ canconfig can0 start
```

- Bring down the device

```
$ ip link set can0 down
```

or

```
$ canconfig can0 stop
```

Set different Bitrate

For setting up the different bitrate form what was set before then follow these steps. In this example bitrate is setting it to 1MBPS

- Bring down the device if it was up

```
$ ip link set can0 down
```

or

```
$ canconfig can0 stop
```

- Set bitrate

```
$ ip link set can0 type can bitrate 1000000 triple-sampling on
```

or

```
$ canconfig can0 bitrate 1000000 ctrlmode triple-sampling on
```

- Bring up the device

```
$ ip link set can0 up
```

or

```
$ canconfig can0 start
```

Test mode

To test the DCAN module then follow these steps

Loopback mode

For testing the module in loopback mode then use below steps

- Bring down the device if it was up

```
$ ip link set can0 down
```

or

```
$ canconfig can0 stop
```

- Configure the loopback mode with triple sampling on and 100KBPS bitrate

```
$ canconfig can0 bitrate 100000 ctrlmode triple-sampling on loopback on
```

- Bring up the device

```
$ ip link set can0 up
```

or

```
$ canconfig can0 start
```

Silent mode

For testing the module in silent mode then use below steps

- Bring down the device if it was up

```
$ ip link set can0 down
```

or

```
$ canconfig can0 stop
```

- Configure the listen-only mode with triple sampling on and 500KBPS bitrate

```
$ canconfig can0 bitrate 500000 ctrlmode triple-sampling on listen-only on
```

- Bring up the device

```
$ ip link set can0 up
```

or

```
$ canconfig can0 start
```

Statistics of CAN

Statistics of CAN device can be seen from these commands

```
$ ip -d -s link show can0
```

Below command also used to know the details

```
$ cat /proc/net/can/stats
```

Error frame details

DCAN IP Error details

If the CAN bus is not properly connected or some hardware issues DCAN has the intelligence to generate an Error interrupt and corresponding error details on hardware registers.

In CAN terminology errors are divided into three categories

- Error warning state, this state is reached if the error count of transmit or receive is more than 96.
- Error passive state, this state is reached if the core still detecting more errors and error counter reaches 127 then bus will enter into
- Bus off state, still seeing the problems then it will go to Bus off mode.

DCAN driver provides

For the above error state, driver will send the error frames to inform that there is error encountered. Frame details with respect to different states are listed here:

- Error warning frame

```
<0x004> [8] 00 08 00 00 00 00 60 00
```

ID for error warning is 0x004 [8] represents 8 bytes have received 0x08 at 2nd byte represents type of error warning. 0x08 for transmission error warning, 0x04 for receive error warning frame 0x60 at 7th byte represent tx error count.

- Error passive frame

```
<0x004> [8] 00 10 00 00 00 00 00 64
```

ID for error passive frame is 0x004 [8] represents 8 bytes have received 0x10 at 2nd byte represents type of error passive. 0x10 for receive error passive, 0x20 for transmission error passive 0x64 at 8th byte represent rx error count.

- Buss off state

```
<0x040> [8] 00 00 00 00 00 00 00 00
```

ID for bus-off state is 0x040

Error frames display with candump

candump has the capability to display the error frames along with data frames on the console. Some of the error frames details are mentioned in the previous section

```
$ candump can0 --error
```

Linux Driver Configuration

- DCAN 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 (details below)

How DCAN driver fits into Linux architecture

- DCAN driver is a can "networking" driver that fits into the Linux Networking framework
- It is available as a configuration item in the Linux kernel configuration as follows:

```
  Linux Kernel Configuration
     Networking support
        CAN bus subsystem support
           CAN device drivers
              Bosch D_CAN devices
                Generic Platform Bus based D_CAN driver
```

Detailed Kernel Configuration

To enable/disable CAN driver support, start the *Linux Kernel Configuration* tool:

```
$ make menuconfig ARCH=arm
```

Select *Networking support* from the main menu.

```
    ...
    ...
    Power management options --->
[*] Networking support --->
    Device Drivers --->
    File systems --->
    Kernel hacking --->
    ...
    ...
```

Select *CAN bus subsystem support* as shown here:

```
    ...
    ...
    Networking options  --->
[ ] Amateur Radio support  --->
<*> CAN bus subsystem support  --->
    IrDA (infrared) subsystem support  --->
    ...
```

Select *Raw CAN Protocol & Broadcast Manager CAN Protocol* as shown here:

```
    ...
    --- CAN bus subsystem support
<*> Raw CAN Protocol (raw access with CAN-ID filtering)
<*> Broadcast Manager CAN Protocol (with content filtering)
    CAN Device Drivers  --->
```

Building D_CAN driver into Kernel

By default D_CAN driver is included in the Kernel

Select *Bosch D_CAN devices* in the above menu and then select the following options:

```
<*> Virtual Local CAN Interface (vcan)
<*> Platform CAN drivers with Netlink support
[*]   CAN bit-timing calculation
< >   TI High End CAN Controller
< >   Microchip MCP251x SPI CAN controllers
< > Philips/NXP SJA1000 devices  --->
< > Bosch C_CAN devices  --->
<*> Bosch D_CAN devices  --->
    CAN USB interfaces  --->
          ...
```

Note: "CAN bit-timing calculation" needs to be enabled to use "ip" utility to set CAN bitrate

Select *Generic Platform Bus based D_CAN driver* as shown here:

```
--- Bosch D_CAN devices
<*>   Generic Platform Bus based D_CAN driver
```

Building D_CAN driver as Loadable Kernel Module

To build the *Bosch D_CAN devices* components as module, press 'M' key after navigating to config entries

```
<*> Virtual Local CAN Interface (vcan)
<*> Platform CAN drivers with Netlink support
[*]   CAN bit-timing calculation
< >   TI High End CAN Controller
< >   Microchip MCP251x SPI CAN controllers
< > Philips/NXP SJA1000 devices  --->
< > Bosch C_CAN devices  --->
<M> Bosch D_CAN devices  --->
    CAN USB interfaces  --->
          ...
```

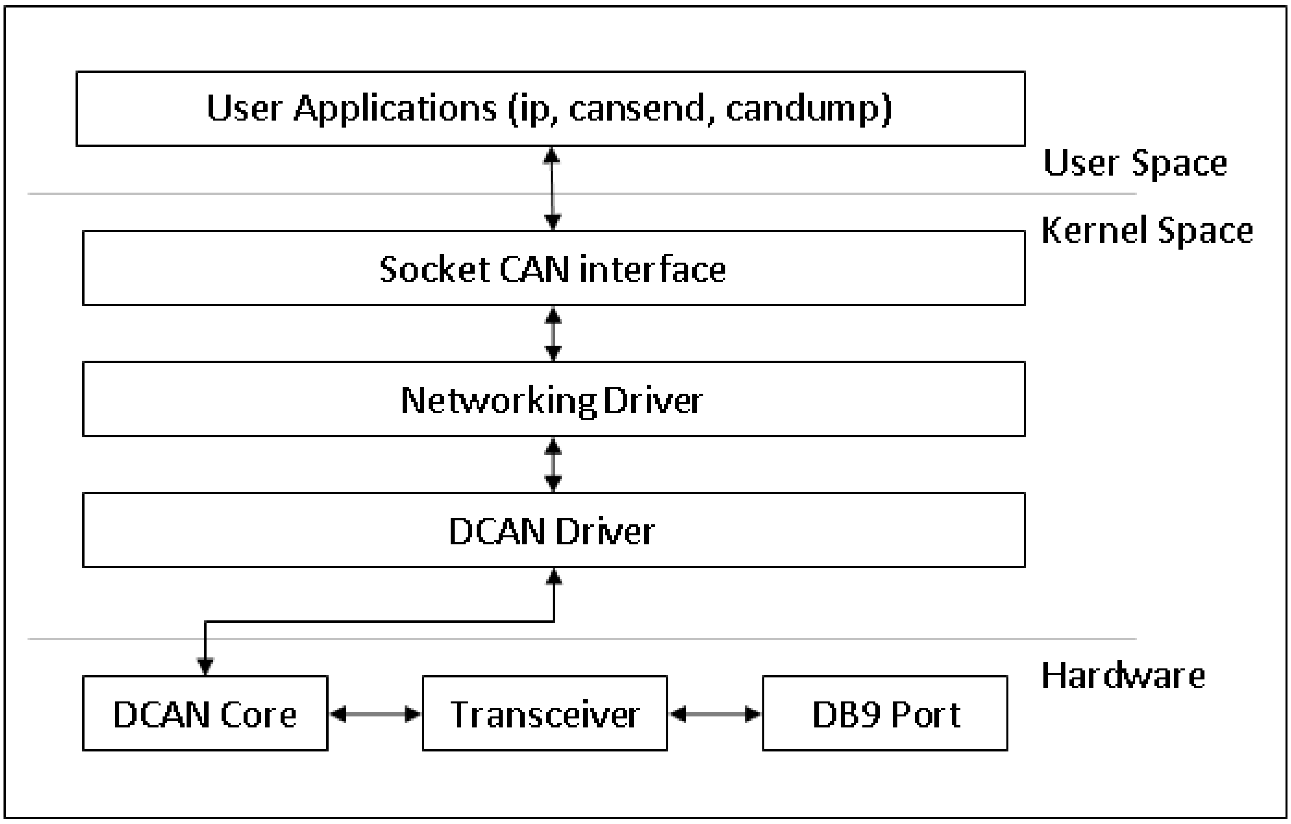
Note: "CAN bit-timing calculation" needs to be enabled to use "ip" utility to set CAN bitrate

To build the *Generic Platform Bus based D_CAN driver* components as module, press 'M' key after navigating to config entries

```
--- Bosch D_CAN devices
<M>   Generic Platform Bus based D_CAN driver
```

DCAN driver Architecture

DCAN driver architecture shown in the figure below, is mainly divided into three layers Viz user space, kernel space and hardware.



User Space

CAN utils are used as the application binaries for transfer/receive frames. These utils are very useful for debugging the driver.

Kernel Space

This layer mainly consists of the socketcan interface, network layer and DCAN driver.

Socketcan interface provides a socket interface to user space applications and which builds upon the Linux network layer. DCAN device 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.

The protocol family module provides an API for transport protocol modules to register, so that any number of transport protocols can be loaded or unloaded dynamically.

In fact, the can core module alone does not provide any protocol and cannot be used without loading at least one additional protocol module. Multiple sockets can be opened at the same time, on different or the same protocol module and they can listen/send frames on different or the same CAN IDs.

Several sockets listening on the same interface for frames with the same CAN ID are all passed the same received matching CAN frames. An application wishing to communicate using a specific transport protocol, e.g. ISO-TP, just selects that protocol when opening the socket. Then can read and write application data byte streams, without having to deal with CAN-IDs, frames, etc.

Hardware

This layer mainly consisting of DCAN core and DCAN IO pins for packet Transmission or reception.

Files

S.No	Location	Description
1	drivers/net/can/d_can/d_can.c	DCAN driver core file
2	drivers/net/can/d_can/d_can_platform.c	Platform DCAN bus driver
3	arch/arm/mach-omap2/board-am335xevm.c	DCAN board specific data addition
4	arch/arm/mach-omap2/devices.c	DCAN soc specific data addition

CAN Utilities

- Since CAN is a "networking" interface and uses the socket layer concepts, many utilities have been developed in open source for utilizing CAN interface.
- For testing CAN we commonly use the cansend /cangen and candump utilities to send and receive packets via CAN module.
- To configure the CAN interface netlink standard utilities are used and this requires iproute2 utilities.

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 then only setting the bitrate is sufficient as it calculates other parameters

Source code

ip (route2)

Source for iproute2 is available at fedora project [4]

canutils

canutils build is depends on libsocketcan binaries so build libsocketcan first then proceed to canutils

Source for canutils are available at pengutronix website [5]

Source for libsocketcan is available at pengutronix website [6]

Build steps

ip cross compilation

- Modifications

Makefile modifications are needed for cross compiling the source for ARM

Comment these lines from top Makefile, and set appropriate environment variables for building

```
- DESTDIR=/usr/
+ #DESTDIR=/usr/
ROOTDIR=$(DESTDIR)
LIBDIR=/usr/lib/
```

```
# Path to db_185.h include
- DBM_INCLUDE:=$(ROOTDIR)/usr/include
+ #DBM_INCLUDE:=$(ROOTDIR)/usr/include
```

```
- CC = gcc
+ #CC = gcc
```

Note: Do not build arpd

```
cp misc/Makefile{,.orig}
sed '/^TARGETS/s@arpd@@g' misc/Makefile.orig > misc/Makefile
```

- Environment variables

Make sure that TOOL CHAIN path (TOOL_CHAIN_PATH) and target file system (FILESYS_PATH) paths are exported along with these

```
export GNUEABI=arm-arago-linux-gnueabi
export CC=$GNUEABI-gcc
export LD=$GNUEABI-ld
export NM=$GNUEABI-nm
export AR=$GNUEABI-ar
export RANLIB=$GNUEABI-ranlib
export CXX=$GNUEABI-c++
export PREFIX=$FILESYS_PATH/usr
export CROSS_COMPILE_PREFIX=$PREFIX
export PATH=$TOOL_CHAIN_PATH/bin:$PATH
export DBM_INCLUDE=/usr/include
export INCLUDES=/usr/include
export DESTDIR=$PREFIX/
```

- Configuration

./configure --host=arm-arago-linux --prefix=$PREFIX --enable-debug

- Build & Install

```
make
sudo make install
```

- Move ip executable to correct directory

Above steps will install binary under $FILESYS_PATH/usr/sbin, rename ip to ip.iproute2 and move it to $FILESYS_PATH/sbin/. folder.

```
mv /usr/sbin/ip /sbin/ip.iproute2
```

libsocketcan cross compilation

- Environment variables

Make sure that TOOL CHAIN path (TOOL_CHAIN_PATH) and target file system (INSTALL_PATH) paths are exported along with these variables. Example INSTALL_PATH is PWD/install (present working directory is LIBSOCKETCAN_PATH).

Note, create "install" directory under LIBSOCKETCAN_PATH.

```
export GNUEABI=arm-arago-linux-gnueabi
export CC=$GNUEABI-gcc
export LD=$GNUEABI-ld
export NM=$GNUEABI-nm
export AR=$GNUEABI-ar
export RANLIB=$GNUEABI-ranlib
export CXX=$GNUEABI-c++
export INSTALL_PATH=$PWD
export PREFIX=$INSTALL_PATH/
export CROSS_COMPILE_PREFIX=$PREFIX
export PATH=$TOOL_CHAIN_PATH/bin:$PATH
```

- Configuration

```
./configure --host=arm-arago-linux --prefix=$PREFIX --enable-debug
```

- Build

```
make
```

- Install

```
make install
```

canutils cross compilation

- Environment variables

Make sure that TOOL CHAIN path (TOOL_CHAIN_PATH) and target file system (FILESYS_PATH) paths are exported along with these. Package configuration path (PKG_CONFIG_PATH) should point to libsocketcan config file which is present under libsocketcan source folder (LIBSOCKETCAN_PATH).

```
export GNUEABI=arm-arago-linux-gnueabi
export CC=$GNUEABI-gcc
export LD=$GNUEABI-ld
export NM=$GNUEABI-nm
export AR=$GNUEABI-ar
export RANLIB=$GNUEABI-ranlib
export CXX=$GNUEABI-c++
export PREFIX=$FILESYS_PATH/usr
export CROSS_COMPILE_PREFIX=$PREFIX
export PATH=$TOOL_CHAIN_PATH/bin:$PATH
export LIBSOCKETCAN_INSTALL_DIR=$LIBSOCKETCAN_PATH/install
export PKG_CONFIG_PATH=$LIBSOCKETCAN_PATH/config
export LD_LIBRARY_PATH=${LIBDIR}:${LD_LIBRARY_PATH}
export LD_RAN_PATH=${LIBDIR}:${LD_RAN_PATH}
export LDFLAGS="-Wl,--rpath -Wl,$LIBSOCKETCAN_INSTALL_DIR/lib"
export INCLUDES="-I$LIBSOCKETCAN_INSTALL_DIR/include"
```

- Configuration

```
./configure --host=arm-arago-linux --prefix=$PREFIX --enable-debug
```

- Build & Install

```
make
sudo make install
```

Generic usage details

ip

- "ip" utility help - ensure you are using iproute2 utility as only that supports CAN interface.

```
$ ./ip link help
```

```
Usage: ip link add link DEV [ name ] NAME
                  [ txqueuelen PACKETS ]
                  [ address LLADDR ]
                  [ broadcast LLADDR ]
                  [ mtu MTU ]
                  type TYPE [ ARGS ]
      ip link delete DEV type TYPE [ ARGS ]

      ip link set { dev DEVICE | group DEVGROUP } [ { up | down } ]
                         [ arp { on | off } ]
                         [ dynamic { on | off } ]
                         [ multicast { on | off } ]
                         [ allmulticast { on | off } ]
                         [ promisc { on | off } ]
                         [ trailers { on | off } ]
                         [ txqueuelen PACKETS ]
                         [ name NEWNAME ]
                         [ address LLADDR ]
                         [ broadcast LLADDR ]
                         [ mtu MTU ]
                         [ netns PID ]
                         [ alias NAME ]
                         [ vf NUM [ mac LLADDR ]
                                  [ vlan VLANID [ qos VLAN-QOS ] ]
                                  [ rate TXRATE ] ]
                         [ master DEVICE ]
                         [ nomaster ]
      ip link show [ DEVICE | group GROUP ]

TYPE := { vlan | veth | vcan | dummy | ifb | macvlan | can }
```

Note: check the result of `ip link help` ;ensure that the `TYPE` field contains `can`

- Ensure you are using the right ip utility (from iproute2)

```
$ ./ip -V
ip utility, iproute2-ss110629
```

cansend

```
$ ./cansend can0 -i 0x10 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88
$ ./cansend can0 -i 0x55 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 -e
```

candump

```
$ candump can0
```

cansequence

```
$ cansequence can0
$ cansequence can0 -p
```

canconfig

```
$ canconfig can0 bitrate 100000
$ canconfig can0 start
$ canconfig can0 stop
$ canconfig can0 ctrlmode loopback on
```

Miscellaneous

Support for second D_CAN instance (AM335x)

On AM335x EVM only D_CAN1 is supported. Hence in `arch/arm/mach-omap2/board-am335xevm.c` only D_CAN1 is support is enabled by default. To add support for D_CAN0 on your custom board, follow these steps.

Add pinmux configurations

Add a `pinmux_config` structure in your custom AM335x board file for instance 0. AM335x SoC provides multiple options for D_CAN0 pins. Refer to the schematic of your board to find which exact D_CAN0 pins are connected to the transceiver. In the example below, we assume pin UART0 RXD (primary function) is used as D_CAN0 TX pin and UART0 TXD (primary function) is used as D_CAN0 RX pin.

```
static struct pinmux_config d_can0_pin_mux[] = {
        {"uart0_rxd.d_can0_tx", OMAP_MUX_MODE2 | AM33XX_PULL_ENBL},
        {"uart0_txd.d_can0_rx", OMAP_MUX_MODE2 | AM33XX_PIN_INPUT_PULLUP},
        {NULL, 0},
};
```

Now, call the `setup_pin_mux()` API to affect these mux configurations. This has to be done inside the `d_can_init()` API in `arch/arm/mach-omap2/board-am335xevm.c`, which is called from the board initialization code.

```
setup_pin_mux(d_can0_pin_mux);
```

Registering platform device

Next, the D_CAN0 platform device needs to be registered. For this, just call `am33xx_d_can_init()` with instance number argument passed as 0. This also needs to be done in the board initialization sequence.

```
am33xx_d_can_init(0);
```

You are now all set to test the second D_CAN instance. Note that the CAN instance which is registered first appears as `can0` irrespective of which instance number it corresponds to. The second CAN instance appears as `can1`.

References

[1] http://search.digikey.com/us/en/products/AK152-2-R/AE9872-ND/821627
[2] http://www.assmann.us/specs/AK152-2-R.pdf
[3] http://processors.wiki.ti.com/index.php/AM335x_General_Purpose_EVM_HW_User_Guide#Configuration.2FSetup
[4] http://pkgs.fedoraproject.org/repo/pkgs/iproute/iproute2-2.6.39.tar.gz/8a3b6bc77c2ecf752284aa4a6fc630a6/iproute2-2.6.39.tar.gz
[5] http://git.pengutronix.de/?p=tools/canutils.git;a=shortlog;h=refs/heads/master
[6] http://www.pengutronix.de/software/libsocketcan/download/libsocketcan-0.0.8.tar.bz2

Article Sources and Contributors

AM335X DCAN Driver Guide *Source*: http://processors.wiki.ti.com/index.php?oldid=127383 *Contributors*: Chanilkumar

Image Sources, Licenses and Contributors

Image:TIBanner.png *Source*: http://processors.wiki.ti.com/index.php?title=File:TIBanner.png *License*: unknown *Contributors*: Nsnehaprabha
Image:am335x dcan cable details.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Am335x_dcan_cable_details.png *License*: unknown *Contributors*: Chanilkumar
Image:am335x dcan port details.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Am335x_dcan_port_details.png *License*: unknown *Contributors*: Chanilkumar
Image:Dcan_driver_architecture.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Dcan_driver_architecture.png *License*: unknown *Contributors*: Chanilkumar