**Background**

The Seeed CAN hat ( <https://www.seeedstudio.com/CAN-BUS-FD-HAT-for-Raspberry-Pi-p-4742.html>) with the Raspberry Pi4 provides two CAN interfaces. The objective is to setup the Rpi software so that it duplicates the functionality that the older approach with a STM32 running a gateway program, with UART-USB interface, with socat to convert from USB-serial to internet socket, with hub-server to allow multiple connections.

The initial os for the Rpi can be setup with the following video (selected amongst many available)--

<https://www.youtube.com/watch?v=BpJCAafw2qE>

**Setup**

**A. /boot/config.txt**

Note: The SD card with the OS for the RPi can be removed from the Rpi and mounted on a PC. Even though the SD card is setup to boot the Rpi the partition with the boot directory will mount on a PC. This makes it possible to edit files on the boot directory.

Note: Additional info is available on /boot/config.txt at https://www.raspberrypi.org/documentation/configuration/config-txt/README.md

Append the following line to--/boot/config.txt

dtoverlay=seeed-can-fd-hat-v2

Uncomment the two lines that enable i2c and spi, so that it looks like the following--

# Uncomment some or all of these to enable the optional hardware interfaces

dtparam=i2c\_arm=on

#dtparam=i2s=on

dtparam=spi=on

**B. /etc/rc.local**

Before the exit 0 in /etc/rc.local, add the following lines--

(Should txqueuelen be set in this too?

sudo ifconfig can0 txquenuelen 4096

sudo ifconfig can1 txquenuelen 4096)

ip link set can0 type can bitrate 500000

ip link set can1 type can bitrate 500000

ip link set up can0

ip link set up can1

# Enable CAN gateway

#

modprobe can-gw

# Creating vcan[01] interfaces

sudo ip link add dev vcan0 type vcan

sudo ip link set up vcan0

sudo ip link add dev vcan1 type vcan

sudo ip link set up vcan1

**C. can-utils**

‘can-utils’ are utility programs for working with the CAN interfaces. These are installed with--

sudo apt install can-utils

(In the above, vcan0 and vcan1 are “virtual” CAN interfaces which could be omitted.)

**D. Check progress**

After rebooting, the CAN interfaces can be checked by executing the following from a terminal window--

ifconfig -a

[The result should be similar to the following]

can0: flags=193<UP,RUNNING,NOARP> mtu 16

unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 txqueuelen 10 (UNSPEC)

RX packets 31413189 bytes 164160310 (156.5 MiB)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 3645 bytes 12581 (12.2 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

device interrupt 66

can1: flags=193<UP,RUNNING,NOARP> mtu 16

unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 txqueuelen 10 (UNSPEC)

RX packets 51652 bytes 65690 (64.1 KiB)

RX errors 8 dropped 0 overruns 0 frame 8

TX packets 11786 bytes 34982 (34.1 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

device interrupt 67

eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500

ether dc:a6:32:43:08:d3 txqueuelen 1000 (Ethernet)

RX packets 0 bytes 0 (0.0 B)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 0 bytes 0 (0.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536

inet 127.0.0.1 netmask 255.0.0.0

inet6 ::1 prefixlen 128 scopeid 0x10<host>

loop txqueuelen 1000 (Local Loopback)

RX packets 492031 bytes 152768883 (145.6 MiB)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 492031 bytes 152768883 (145.6 MiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

vcan0: flags=193<UP,RUNNING,NOARP> mtu 72

unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 txqueuelen 1000 (UNSPEC)

RX packets 866 bytes 4330 (4.2 KiB)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 52788 bytes 259881 (253.7 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

vcan1: flags=193<UP,RUNNING,NOARP> mtu 72

unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 txqueuelen 1000 (UNSPEC)

RX packets 226 bytes 1130 (1.1 KiB)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 226 bytes 1130 (1.1 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500

inet 192.168.2.139 netmask 255.255.255.0 broadcast 192.168.2.255

inet6 fe80::7612:6d0b:211d:4457 prefixlen 64 scopeid 0x20<link>

ether dc:a6:32:43:08:d4 txqueuelen 1000 (Ethernet)

RX packets 24998328 bytes 1305526222 (1.2 GiB)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 25106324 bytes 3124739608 (2.9 GiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

In the above one can see the presence of the two harward CAN interfaces plus the two virtual interfaces.

**E. Programs**

Load repository from github into GliderWinchItems directory--

cd

git clone ssh://git@github.com/GliderWinchItems/can-server

Compile programs--

cd ~/GliderWinchItems/can-server

make

Three programs should compile with the single make of Makefile--

can-server

can-client

can-bridge

**can-server**

This routine is hack of socketcand which is located in the github repo linux-can/can-utils.

(Note: the name was changed to can-server since the apt-get install can-utils, above, installed socketcand which is the non-hacked routine and uses a command line command system and different CAN msg formats. If the names were the same, executing ./socketcand and socketcand would get different programs and therefore a source of confusion.)

The program flow of can-server begins with parsing the command line parameters. When there are errors the program exits. When it completes successfully there is “break” in the “for ( ;; )” loop and the next phase handles the address, port and opening of a listening socket.

The program then waits for an incoming connection. When that arrives a socket for that connection is setup and a fork() is executed. The main program loops back and listens for another connection. The child process from the fork opens a raw socket for the CAN interface specified on the command line. The child process then enters an endless loop.

The child process endless loop executes a select that waits for available input on either the internet socket for the connection, or the raw socket for the CAN interface. When select exits the wait ‘if’s determine which, or both, had data.

The incoming CAN data arrives in a struct with the CAN id, dlc, and data in “their” format. The data from the struct is revised and and converted to ascii to “our” format, and a send is made to the internet socket.

The struct is defined in the directory /usr/include/linux, and file can.h, reproduced here--

/\*\*

\* struct can\_frame - basic CAN frame structure

\* @can\_id: CAN ID of the frame and CAN\_\*\_FLAG flags, see canid\_t definition

\* @can\_dlc: frame payload length in byte (0 .. 8) aka data length code

\* N.B. the DLC field from ISO 11898-1 Chapter 8.4.2.3 has a 1:1

\* mapping of the 'data length code' to the real payload length

\* @\_\_pad: padding

\* @\_\_res0: reserved / padding

\* @\_\_res1: reserved / padding

\* @data: CAN frame payload (up to 8 byte)

\*/

struct can\_frame {

canid\_t can\_id; /\* 32 bit CAN\_ID + EFF/RTR/ERR flags \*/

\_\_u8 can\_dlc; /\* frame payload length in byte (0 .. CAN\_MAX\_DLEN) \*/

\_\_u8 \_\_pad; /\* padding \*/

\_\_u8 \_\_res0; /\* reserved / padding \*/

\_\_u8 \_\_res1; /\* reserved / padding \*/

\_\_u8 data[CAN\_MAX\_DLEN] \_\_attribute\_\_((aligned(8)));

};

The incoming msg from the internet socket arrives as a chars in a stream. The routine can-so.c (Seeed/socket to Old/our format conversion) extracts ‘\n’ terminated lines from the incoming stream and converts them to the struct format used by the CAN raw socket. Error checking is applied. The expected ascii-hex line format for the CAN msg is the ascii-hex of the following binary (and terminated with a ‘\n’)--

byte 1 sequence number

byte 2 low order byte of 32b STM32 format CAN id

byte 3,4 ...

byte 5 high order byte of CAN id

byte 6 dlc (length of payload)

byte 7-14 (variable length payload bytes)

last byte – checksum of above (binary)

When a send to the CAN raw socket is made, the underlying code sends a copy to all the other connections to that CAN interface. The result is that an incoming ascii-hex msg from one of the internet socket connections gets sent to all the other connections as well as the CAN interface. This is all done at the kernel/driver level.

The following starts can-server:

cd ~/GliderWinchItems/can-server

./can-server ./can-server -l wlan0 -i can1 -p 29537 -v

Where:

-l wlan0 = Rpi wifi (see ifconfig output above)

-i can1 = CAN interface: either can0 or can1

-p 29537 = listening port (default is 29536)

-v = verbose output

**can-client**

can-client is the same program as can-server except for the internet socket setup. can-client expects to connect to a server, whereas can-server waits for clients to connect.

Since programs such as cangate, cangateCON, netcat, make client type connections, they have to work with a server, e.g. can-server, or hub-server. To be practical, can-client would need to work with an instance of hub-server. In such a configuration can-client the arrangement would be similar to socat, that converts usb-serial to an internet socket that then connects to hub-server. With can-server it all takes place in the same program, and without tcp delays.

The following starts hub-server; can-client connects to hub-server; nc displays hub-server lines--

deh@rpi41-deh:~$ hub-server localhost 32127 &

[1] 898

deh@rpi41-deh:~$ nc localhost 32127

[in another terminal window]

deh@rpi41-deh:~/GliderWinchItems/can-server$ ./can-client -s localhost -i can1 -p 32127 -v

**can-bridge & cangw**

The purpose of ‘can-bridge’ is to bridge/gateway CAN msgs from can0 to can1, and vice-versa. This routine runs, but does not implement any msg filtering, i.e. it passes all msgs between the two CAN buses.

‘cangw’ implements this bridge/gateway function, so ‘can-bridge’ is likely not needed.

The command line arguments for cangw can be found in the ‘man’ pages. (They are terse)--

man cangw

The following command adds (-A) a “rule” that will pass all CAN msgs from the source (-s) CAN0 to the destination (-d) CAN1. The msgs passed from CAN0 to CAN1 are “looped back” (-e) so that they appear in the internet output. Since the filter statement (-f) is not included it defaults to “all”.

sudo cangw -A -s can0 -d can1 -e

The rule is remains after cangw exits. To list the rules in effect--

sudo cangw -L

cangw -A -s can0 -d can1 -e # 77415 handled 0 dropped 0 deleted

To clear/flush the rules--

sudo cangw -F

Setting the gateway filtering is done with the following format--

-f <CANid>:<mask> [match when received CANid & mask = rule CANid & mask]

-f <CANid>!<mask> [match when received CANid & mask != rule CANid & mask]

Note that the CAN id is in the CAN socket format:

See can.h in /usr/include/linux directory--

/\*

\* Controller Area Network Identifier structure

\*

\* bit 0-28 : CAN identifier (11/29 bit)

\* bit 29 : error message frame flag (0 = data frame, 1 = error message)

\* bit 30 : remote transmission request flag (1 = rtr frame)

\* bit 31 : frame format flag (0 = standard 11 bit, 1 = extended 29 bit)

\*/

E.g., the following will pass the 64/sec time msgs that have the left justified (STM32 format), and a CANid of 0x00400000, from can0 to can1. (Note: leading zero’s are not needed.)

sudo cangw -A -s can0 -d can1 -e -f 2:1fffffff

Additional resources--

Covers boards using the MCP251x series CAN chips--

<https://developer.ridgerun.com/wiki/index.php/How_to_configure_and_use_CAN_bus>