SIMPLE CAN BUS WITH GO

This project is a collection of Go and C libraries and programs that set up a very simple simulation of a set of devices. It might be useful for someone experimenting with CAN bus with Go for the first time.

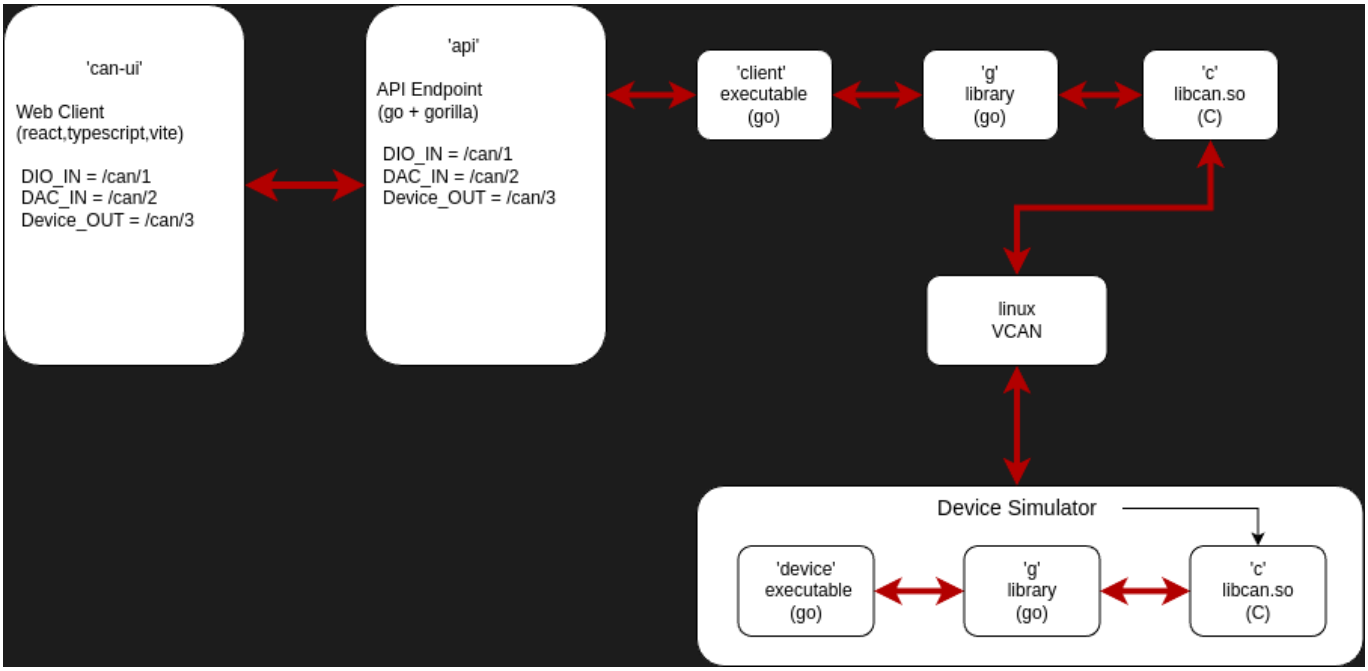
References

The primary sources I used for figuring out CAN and vcan.

https://github.com/linux-can/can-utils https://docs.kernel.org/networking/can.html https://elinux.org/CAN_Bus https://www.pragmaticlinux.com/2021/10/how-to-create-a-virtual-can-interface-on-linux/

CAN BUS SIMULATION

The architecture looks something like this:



Starting at the bottom right of the diagram and going counter-clockwise:

- Device Simulator : simple simulated device accessible over CAN
    - vcan : the standard vcan module
    - libcan.so : C library that performs the setup and connections to the CAN bus interface. The low level incantations to access the CAN bus interface is easier to do in C.
        - in directory "c"
    - can.go : a Go package that uses libcan.so and exposes a Go interface to the CAN bus
        - in directory "g"
    - device : executable device simulator
        - in directory "device"
- CAN bus 'client' : executable that exposes a run time interface to the simulated device.

    - uses vcan, c/libcan.so and g/can.go
    - client : a client side executable program that sends and receives CAN bus messages to and from the device simulator. It exposes a Go interface that allows other apps to send and receive CAN bus messages.
        - in directory "client"
    - api : a Go web API that talks to the 'client' and lets remote programs read and update the state of the CAN bus simulator
        - in directory "api"

- Remote user interface

    - can-ui : a React app that uses the 'api' to provide a user interface to access the CAN bus simuilator
        - in directory "can-ui"

The CAN bus messages include:

- 1 : ID_DIO_IN : the simulated device listens for this message to set a digital IO register.
- 2 : ID_DIO_OUT : the simulated device sends data from a digital IO register at 10 Hz. Applications can listen for this message to get updates.
- 3 : ID_DAC_IN : the simulated device listens for this message to set a digital-to-analog input.
- 4 : ID_ADC_OUT : the simulated devices sends data from an analog-to-digital device at 10 hz.

How To Run

- Install the Linux package 'can-utils. Command line tools that can be used to send and receive CAN messages. Its useful in testing.
- Vcan (Virtual CAN Bus) is a kernel module that creates a virtual CAN bus interface on jthe LINUX system. Install the vcan module on Linux if its not already there. I used Ubuntu 20 which has the support. The instructions here are for debian based systems.
- Install build-essential, golang (18 or later) and nodejs.
- Activate the VCAN module for use as a network device (see below)
- Clone this repo at https://github.com/dmh2000/simple-can-bus
- cd into top level
- execute 'make'
    - the make process is set up to use the local instance of c/libcan.so using LD_LIBRARY_PATH where needed. If you want to make it permanent you can install c/libcan.so into /usr/local/lib. See c/Makefile and uncomment the lines that install that library.
- in a terminal, run the vcan/device/device program
- in a terminal, run the vcan/api/api program
- in a terminal, run the vcan/client/client program
- in a terminal, start the vcan/can-ui web client
    - this requires some node setup. See below for details.

Dependencies

- Linux package 'build-essential;
- node.js

This document assumes a Debian type system, such as Ubuntu. Some operations may be different on a Fedora based system.

Step 1 : Activate the VCAN IP Device

- this step attempts to load the VCAN kernel module
- https://www.pragmaticlinux.com/2021/10/how-to-create-a-virtual-can-interface-on-linux/
- If the modprobe fails to find vcan, then your kernel probably doesn't have CAN support built in so you would have to build a kernel. That's not trivial but its dooable.

```
#!/bin/bash
# Load the kernel module.
sudo modprobe vcan
# Create the virtual CAN interface.
sudo ip link add dev vcan0 type vcan
# Bring the virtual CAN interface online.
sudo ip link set up vcan0
```

Step 2 : Install Linux CAN Utils

https://github.com/linux-can/can-utils

The standard package 'can-utils' includes a bunch of utilities for working with CAN bus on Linux. Using candump or cansend is a good way to test the complement functions in the c or go versions.

```
# INSTALL
sudo apt install can-utils

# TEST
# terminal 1
candump -tz vcan0

# terminal 2
cansend vcan0 123#00FFAA5501020304
```

Step 3 : Review the Source Directories

Refer to the block diagram for the usage and location of the various components.

The first two directories, ./c and ./g build the infrastructure required to use Go to access a CAN bus device. In this case the VCAN simulated device. But it would apply to any conventional CAN device.

./c/canlib.c

See ./c/README.md

Directory 'c' contains a set of stripped down functions that can be used to send and receive data from a CAN bus interface. The code here has the more complicated incantations to connect to a CAN device using the socket interface. This is based on excerpts from https://github.com/linux-can/can-utils

The Makefile builds libcan.so. It has an option to load it to /usr/local/lib which requires sudo permission. Just remove the comments in the libcan.so rules. If you prefer to just keep it local, no change is needed.

Two test programs can_test_receive.c and can_test_send.c exercise the interface.

./g/can.go

See ./g/README.md

Directory 'g' provides a Go module with functions that use the C libcan.so to interface to a CAN bus interface. This module sends and receives **raw CAN frames**. This directory also contains unit tests. The code here intends to be a very simple wrapper over the C library. This provides basic CAN send/receive support for Go. A real CAN device may have more functionality like error handling and filtering incoming messages. Setting that up would depend on the specific device.

The Makefile compiles the module.

./device/device.go

See ./device/README.md

Directory ./device implements a program that simulates a CAN sensor unit. It reads CAN messages from a client program, parses them and sends formattted simulated sensor data to the can bus. Think of this program as the microcomputer sensor sitting in the hardware.

The Makefile compiles to an executable.

Test functions are included using the Go test framework.

The next directories implement the application components needed to implement the example system shown in the diagram.

./client/client.go

See ./client/README.md

Directory ./client provides a Go API accessible by other Go programs to communicate with the 'device' over the CAN bus. The 'client' is not directly connected to the 'device'. All communication between the two is over the CAN bus. The client implements the specified set of inputs and outputs that the device provides. Think of this program as the application 'backend' that sits in some computer in the system.

./api/api.go

See ./api/README.md

Directory "api" contains a web backend that connects to the 'client' program. It provides send/receive functions for a web application that displays and controls the CAN simulation.

./can-ui

See ./can-ui/README.md

A web client for accessing the CAN bus data. Its a React/Vite app. In development mode there is no build operation. Just run it from the command line.

Step 4 : Build the System

The top level Makefile will build all the system components and run the relevant tests. Just execute 'make' at the top level to build.

It builds in this order:

- ./c/Makefile
- ./g/Makefile
- ./device/Makefile
- ./client/Makefile
- ./api/Makefile

The Makefile also includes a 'clean' step.

Step 5 : Run the System

To run the components in separate terminals:

- in a terminal, execute ./device/device
- in a terminal, execute ./api/api
- in a terminal, start the ./can-ui web client
    - cd into that directory and execute:
        - npm install
        - npm run dev

```
gnome-terminal --tab --  ./device/device
gnome-terminal --tab --  ./api/api
pushd ./can-ui
npm install
npm run dev
```