CAN Bus For Software Developers

Controller Area Network, aka CAN, is pretty common these days in embedded systems. It's a pretty good mechanism for passing short messages between devices at a reasonably high rate. It dominates in automotive applications for monitor and control of systems in vehicles.

This article intends to give you an overview of CAN from a pure software view, no hardware.

There are a lot of resources for learning to use a CAN bus system. As a software engineer, I found that most of them have a bit more hardware/electronic focus and not quite enough software 'how-to' on a target system. There's a lot of waveform, timing, electrical stuff that is worth knowing something about but that's transparent to pure software usage.

If you are familiar with the ISO 7-layer model (of which 2 layers are never used in modern system), this discussion is about the **transport layer**. If you are interested in the physical, data link or network layer (doesn't really have one of these), see the references at the end of the article.

CAN Bus Is About Messages, Not Nodes

A CAN bus system doesn't look like a network where nodes are the important things. CAN bus is about messages. Just imagine a cloud where messages appear and are read and acted on. It doesn't care where they came from and where they went.

When you architect a CAN bus system, you can do the whole thing with a graph of messages and associated state machines, without every specifying the physical nodes/CPUs in the system.

Resume beating a dead horse...

CAN Is Pub/Sub

- CAN is a pub/sub architecture.
    - it’s not a master/slave, or client/server.
- It’s a bus. Connected to the bus are various devices and computers. Call them nodes (but don't obsess about nodes).
    - Nodes can 'publish' messages to the bus at any time.
    - Nodes can 'subscribe' for messages on the bus and read them when they appear on the bus.
- Message format is whatever you can fit in 8 bytes.

The Cool Message Trick

All messages have an ID. Once a message is on the bus, its anonymous. Who sent it and who received it is not part of the transport information.

Depending on the CAN version, standard or extended, the message ID is either 11 or 29 bits. **The key trick is that IDs are unique across sending nodes, and the ID determines message priority**. The lower the ID value, the higher the priority.

Important: If a particular node publishes a message with ID 1, then no other node can use that ID. No two nodes can publish messages with the same ID's, EVER. (If they do the bus will be confused and won't work

right)

At the software level, nodes can freely publish messages at any time. There's no coordination between nodes **in software**. Arbitration for the bus is handled in the datalink layer, and it’s very ingenious how that is handled. In the hardware, if two nodes publish messages at the same time (remember the IDs are required to be different), The hardware will let the message with the lowest ID grab the bus and transmit the message. The unit with the larger ID will wait until that message is transmitted and will try again until it succeeds. It's all handled in hardware by the CAN controllers using the ID bits to arbitrate the bus.

Instead of me trying to describe the hardware, take a look at this reference, chapter 4. Or google 'CAN bus arbitration'.

The result is that messages flow according to the message ID's. When you architect a CAN system, you will probably have a spreadsheet that allocates outgoing messages to nodes in the system. Fields in the spreadsheet might have message ID (unique), message rate, and other stuff like description etc. You could have a separate page with who is listening for what message.

CAN controllers

The above description is the basics for software use. These days CAN controller chips have a lot of functionally such as error handling, and filtering messages. For example, a TI CAN controller 4550 datasheet is 150 pages long. But it’s pretty simple to set up a basic CAN bus system in Linux if the kernel is configured for it.

Higher level protocols

In the simplest Ad-Hoc CAN system, you will assign message IDs to the various components in your system and specify all the message formats. If you are using off-the-shelf CAN devices, you will comply with their spec.

There are many higher-level CAN bus protocols, specified by various organizations.

- CANopen
    - used in automation. This is a big fat spec that tries to turn CAN bus into a node-based system.
- Automotive CAN specs
    - SAE J1939
    - ISO 11898 series

References

- TI CAN bus overview
    - https://www.ti.com/lit/an/sloa101b/sloa101b.pdf
- Example TI CAN bus controller
    - https://www.ti.com/lit/ds/symlink/tcan4550-q1.pdf
- Wikipedia of course
    - https://en.wikipedia.org/wiki/CAN_bus

Coming up in next post

The next post will show you how to set up a virtual CAN bus for Linux. It will include a couple of simple libraries that support access to a CAN bus (virtual or real). The libs are in C and Go.