4060 - T29: Real Time Networking
Student Log

Andrew Marinic - 7675509

January - April 2024

Contents

1 Introduction

Intro The following is a log or perhaps a journal of my efforts in COMP 4060 - T29 - Real Time Networking. I will break the document into sections for each Month and subsections for each week. Each day will contain a paragraph explaining what I had accomplished.

2 January

2.1 January 8-12

Jan 8 I did some preliminary reading of the documents provided on CAN networks. I only spent about an hour reading, little notes.

Jan 11 Dug into the data sheets. Started examining the UART drivers we need to build. Probably 30 mins or so taken aside. Industrial Project has hit the fan a bit and taken up more time. I set up a calendar for the group as we all have fairly tight schedules right now.

2.2 January 15-19

Jan 15 We had an hour-long group meeting today. We discussed the pace of the project we would like to achieve. Send off an email to confirm some items and inform the Supervisor of our meeting availability. I started to dig into blinky again with the updated code. To my grief, I have switched to VScode and started to try and get the debugging working with the help of the other group members. I spent another hour or so getting the debugging to work in VScode, I stopped once I had the debugger open, openOCD via VScode. I started to get a little frustrated and had to finish my industrial project proposal so I stopped for the evening. I will discuss with the group for advice and continue later this week.

Jan 17 Today I spent around 4 hours deep-diving CAN and making notes on it. Then I proceeded to try and get debugging working one last time before asking for help. If I cannot accomplish this by, Friday I will reach out. I spent approx 1.5 hours on this, but more time was spent reading about it than coding.

2.2.1 Jan 17 Notes From wiki

Jan 18 Spent approximately 1 hour reading into the Curiosity Nano's CAN controller. I think I understand the requirements for setting up and how the INIT and CONFIG registers work together.

JAN 22 I spent about an hour and a half finishing my CAN notes which are visible in the Notes section. I have done some looking into coding the CAN controller of the curiosity nano.

- https://github.com/MikroElektronika/mikrosdk_click_v2/tree/master/clicks/canfd3 Here is the CAN FD 3 example code. I gave it a quick browse for something valuable but was mainly looking for CAN controller init code keeping it in my back pocket though
- https://microchip-mplab-harmony.github.io/reference_apps/apps/sam_e51_cnano/same51n_mikroe_click/readme.html I also gave themse projects a quick brows for the same as above, but I don't think any use the CAN controller just UART.
- https://microchip-mplab-harmony.github.io/reference_apps/apps/sam_e54_xpro/same54_can_usb_bridge/readme.html#hardware-used I do however think there is something useful here. It uses the same micro controller so it should have INIT, read/write code, ect. This will take some digging though. Today I put in approximately 3.5 hours into researching and documenting.

3 NOTES

3.1 CAN Overview

Controller Area Network (CAN) is a standard for micro-controller and device communication. It uses messages, and was originally designed as a multiplexing method. The physical layer utilises twisted pairs (CAN+ and CAN-). Messages are framed with IDs which dictate priority. Logical 0 is Dominant and logical 1 is recessive. This means IDs that have larger values (1's in high bit places) are lower priority. Arbitration is done via first bit with a 1. All nodes see transmission.

Nodes All nodes can send and receive, but not at the same time. The priority is determined by the frame ID. Messages are transmitted using non-return-to-zero (NRZ) format
All nodes require the following:

- Some controller :CPU/Microprocessor/host processor/micro controller
  - Decides what messages mean and what to transmit
  - Handles talking to other devices
- CAN Controller
  - Receiving: Stores bits until entire message is received, then can trigger interrupt for retrieval
  - Transmitting: If the host, can send messages via CAN controller in a serial manner.
- Transceiver (ISO 11898-2/3) Medium Access Unit (MAU)
  - Receiving: converts at the CAN bus level for CAN controller use, protective layer for CAN controller.
  - Transmitting: converts bit stream from CAN controller to the CAN bus

3.2 CAN: IDs and arbitration

ID Arbitration When nodes transmit they see all messages including their own. If they transmit a 1 and see 0 they quit and lose arbitration. This is because 0 is dominant and they then know they do not have priority. Because of this, whenever there is a collision the lower ID will win. When a collision does happen. The recessive message waits for the dominant message + 6-bit clocks then attempts again This means the first frame to transmit a 1 is the loser, thus highest priority id frame is all 0s followed by 00....001

IDs as priorities Using ID for the type of data, or the sending node ignores the fact ID is also used as message priority. This leads to poor real-time performance. CAN bus is limited to around 30% to ensure deadlines if you don't build around the priority. Otherwise, you can achieve 70 to 80% CAN bus usage and have reliable deadlines.

3.3 CAN: Bit timing

Nominal Bit Time: Time it takes to send bit components:

Synchronization Synchronization is important to the CAN protocol. It prevents errors and allows for arbitration's to occur. Recolonization occurs every single recessive to dominant transmission during the frame. In order to sync the nominal bit time is segmented into quanta and then certain aspects can be altered to allow for synchronization. The nominal bit time is broken down in the following way. Each are assigned a number of quanta. For example a system where we break our nominal bit time into 10 quanta.
Sync (1 quanta)
Propagation (3 quanta)
Phase segment 1 (3 quanta)
Phase segment 2 (3 quanta)
Synchronization occurs as follows.

1. **Bus Idle** -> wait for first recessive to dominant transition
2. **Hard synchronization**
3. **Resync** occurs on every recessive to dominant transition during the frame (message?)
   a. CAN controller expects this at multiple of nominal bit time.
   b. Else It adjust nominal bit time accordingly.

Resync and Adjustment process:

- Produce a number of quanta to divide the bits' segments into time slices.
  - The number of quanta can vary based on the controller
  - The quantity of quanta a segment is assigned can vary depending on system needs
- On out-of-sync (before or after) transition controller calculates the time difference, to compensate:
  - If we need to lengthen we do so to phase 1
  - If we need to reduce time we do so in phase 2.
- As a result of either a or b, we have adjusted the timing of the receiver to the transmitting node and synchronized them.

- We continuously do so at every recessive to dominant transition to keep synchronization
  - This reduces errors induced by noise (random error)
  - Allows for resync to nodes that lost arbitration back to the one that won previously.
  -

3.4 CAN Protocol Layers:

- Application layer
- Object layer
- Transfer layer
- Physical layer

We are building the transfer layer?

3.4.1 CAN Transfer layer:

- Most of the CAN standard applies to this layer, it is what receives messages from the physical layer and into the object layer for use in the application.
- Transfer layer is responsible for:
  - Synchronization
  - Bit timing
  - Message framing
  - Arbitration
  - Acknowledgement
  - Error Detection
  - Signalling
  - Confinement
- To accomplish the previous responsibilities, it performs the following tasks:
  - Fault confinement
  - Error detection
  - Message Validation
  - Arbitration
  - Message framing

  - Transfer rate and timing
  - Information routing

Physical layer

- Pinout:
  - Pin 2: CAN- (Low)
  - Pin 3: GND
  - Pin 7: CAN+ (High)
  - Pin 9: CAN V+ (power)

3.5 CAN Frames:

- Two types of frame format
  1. Base frame format
     i. 11- bits for identifier
     ii. IDE bit dominant
  2. Extended frame format
     i. 11- bit identifier + 18-bit extension = 29-bit identifier
     ii. IDE bit recessive
- There are four types of frames.
- Regardless of type all begin with a start of frame (SOF) bit to signal start of frame transmission.
- Frame types:
  1. Data Frame: a frame containing node data for transmission .
  2. Remote frame: requests transmission of an identifier
  3. Error frame: frame type for any node detecting an error.
  4. Overload frame: a buffer/delay for data or remote frame.

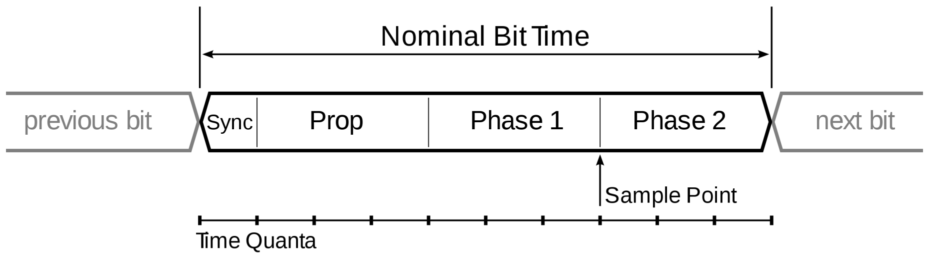
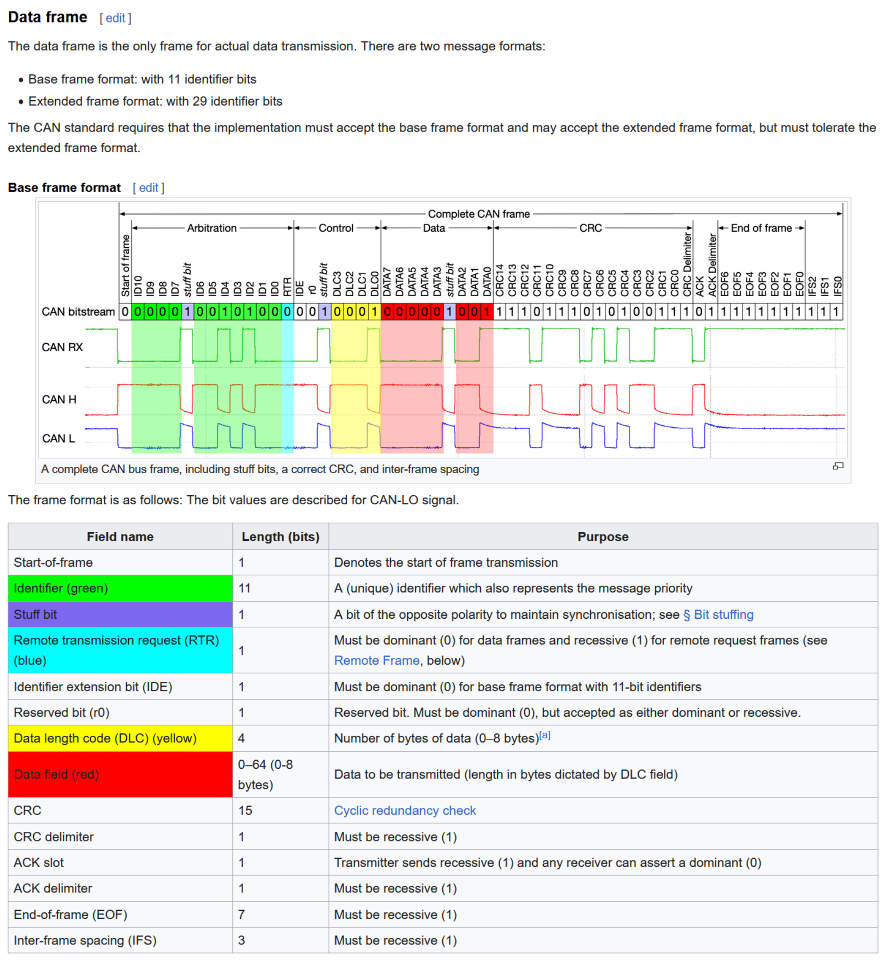


Figure 1: CAN Bit Timing: Wikipedia



Data frame [edit]

The data frame is the only frame for actual data transmission. There are two message formats:

- Base frame format: with 11 identifier bits
- Extended frame format: with 29 identifier bits

The CAN standard requires that the implementation must accept the base frame format and may accept the extended frame format, but must tolerate the extended frame format.

Base frame format [edit]

Complete CAN frame
Arbitration
Control
Data
CRC
End of frame
Start of frame ID10 ID9 ID8 ID7 stuff bit ID6 ID5 ID4 ID3 ID2 ID1 ID0 RTR IDE r0 stuff bit DLC3 DLC2 DLC1 DLC0 DATA7 DATA6 DATA5 DATA4 DATA3 stuff bit DATA2 DATA1 DATA0 CRC14 CRC13 CRC12 CRC11 CRC10 CRC9 CRC8 CRC7 CRC6 CRC5 CRC4 CRC3 CRC2 CRC1 CRC0 CRC Delimiter ACK ACK Delimiter EOF6 EOF5 EOF4 EOF3 EOF2 EOF1 EOF0 IFS2 IFS1 IFS0
CAN bitstream 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 1 1 1 0 1 1 1 0 1 0 1 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
CAN RX
CAN H
CAN L

A complete CAN bus frame, including stuff bits, a correct CRC, and inter-frame spacing

The frame format is as follows: The bit values are described for CAN-LO signal.

Field name	Length (bits)	Purpose
Start-of-frame	1	Denotes the start of frame transmission
Identifier (green)	11	A (unique) identifier which also represents the message priority
Stuff bit	1	A bit of the opposite polarity to maintain synchronisation; see § Bit stuffing
Remote transmission request (RTR) (blue)	1	Must be dominant (0) for data frames and recessive (1) for remote request frames (see Remote Frame, below)
Identifier extension bit (IDE)	1	Must be dominant (0) for base frame format with 11-bit identifiers
Reserved bit (r0)	1	Reserved bit. Must be dominant (0), but accepted as either dominant or recessive.
Data length code (DLC) (yellow)	4	Number of bytes of data (0–8 bytes)[a]
Data field (red)	0–64 (0-8 bytes)	Data to be transmitted (length in bytes dictated by DLC field)
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be recessive (1)
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK delimiter	1	Must be recessive (1)
End-of-frame (EOF)	7	Must be recessive (1)
Inter-frame spacing (IFS)	3	Must be recessive (1)

Figure 2: Data frame: Wikipedia