Network management Assignment2

李俊德，P76074478，蕭丞志，Q36074316
王昱翔，Q36074154，蘇致翰，P76074070

Introduction of protocol

1. History

於 1983 年被開發，在 1986 年，於美國密西根州底特律市舉行的汽車工程師學會（SAE）會議上正式發表。這個協定會被發展出來的主要原因是為了取代傳統佈線直連，採用複用通訊來降低成本與複雜度。為了解決這個問題，can bus 協定有以下特點

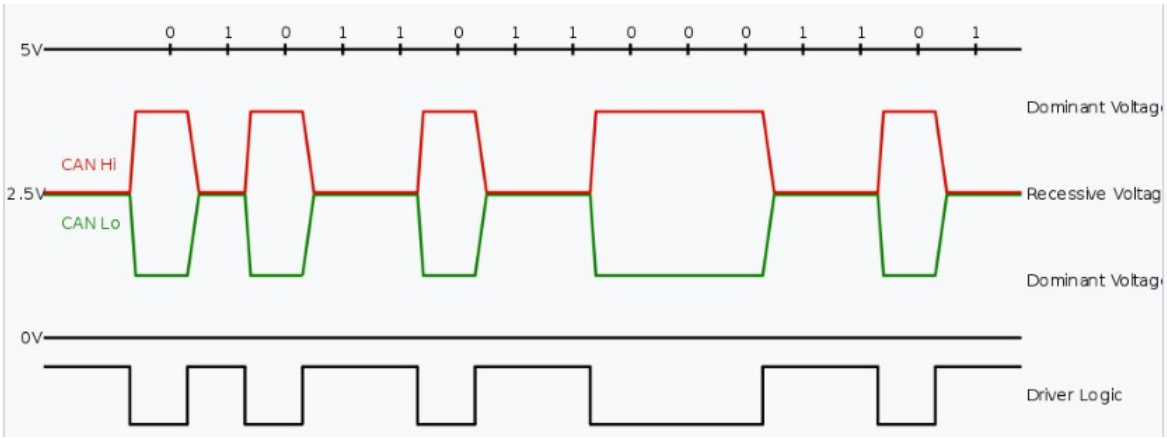
1. 在不需要主機的情況上，允許網路上各個節點(node)間的通訊
2. CAN 建立在基於資訊導向傳輸協定的廣播機制上，利用 Message Identifier 來決定訊息傳遞優先順序，而不是指派特定位址

2. Applications

1. 汽車(起初發展是為了應用在汽車上)、交通工具
2. 航空、航海電子儀器
3. 工業自動化及機械控制

3. Message format

1. CAN bus 協定利用兩條電線訊號來表示邏輯訊號，方法如下



當一條電線訊號被驅動到高電位，一條電線被驅動到低電位時，表示邏輯訊號 0。當兩條電線都沒有被驅動時，表示邏輯訊號 1。

2. 所以這個協定定義邏輯訊號 0 稱為顯性(被驅動)，邏輯訊號 1 稱為隱性(沒被驅動)
3. 每一個封包(基本影格)的訊息內容，以下用邏輯訊號表示

欄位名	字長（位）	作用
起始位（SOF）	1	表示影格的傳輸開始
標誌符（ID\green）	11	唯一辨識碼，同樣代表了優先級
遠端傳輸請求（RTR\藍色）	1	資料框時一定是顯性（0），遠端請求影格時一定是隱性（1）（詳見遠端影格章節）
標誌符拓展位（IDE）	1	對於只有11位標誌符的基本影格格式，此段一定位顯性（0）
預留位（R0）	1	預留位一定是顯性（0），但是隱性（1）同樣是可接受的
資料長度程式碼（DLC\黃色）	4	資料的位元組數（0-8位元組）[a]
資料段（Data field\紅色）	0–64 (0-8 位元組)	待傳輸資料（長度由資料長度碼DLC指定）
迴圈冗餘校驗（CRC）	15	循環冗餘校驗
迴圈冗餘校驗定界符	1	一定是隱性（1）
確認槽（ACK）	1	發信器傳送隱性（1）但是任何接收器可以宣示顯性（0）
確認定界符（ACK delimiter）	1	一定是隱性（1）
結束位(EOF)	7	一定是隱性（1）

4. 以上的影格結構可以對應到實做部份的 typedefine.h，這邊值得注意的是 ID、DLC、Data 的欄位。ID 不可重複，而且 ID 越低，優先度越高，後面會講解仲裁機制。DLC 代表的是這個封包內的資料有多少 byte，雖然 4 bits 可以存 0~15，但根據協定，資料最長只能是 8 bytes。Data field 的內容就是資料，最少為 0 byte，最長為 8 bytes。

4. 仲裁機制

當同時有兩個節點想要傳送資料時，會採用無失真位仲裁解決辦法。在舉例講解之前有一個前提要先說，就是「CAN bus 在傳送資料時要求 CAN 網路上對每一位的採樣都在同一時間」。接下來講解仲裁機制如下，假設同一時間有兩個節點想要傳輸資料，他們會直到兩個人傳送不同訊號的時候才發現衝突，這時候會是顯性位獲勝。舉個例子來說：現在 ID:15(00000001111)和 ID:16(00000010000)同時傳輸，如下圖

	起始位	ID位										影格剩下的部分
		10	9	8	7	6	5	4	3	2	1	0	
節點15	0	0	0	0	0	0	0	0	1	1	1	1	
節點16	0	0	0	0	0	0	0	1	停止傳輸				
CAN資料	0	0	0	0	0	0	0	0	1	1	1	1	

由此例子也可以知道為什麼 ID 越小優先度越高，以及為何不能有相同的 ID，這樣當衝突發生時，兩個節點都會認為自己正常的傳輸訊息。

5. Error checking

1. CRC

   CRC 在訊息結尾處加上一個 FCS（frame check sequence）來確保訊息的正確。接收訊息端會將其 FCS 重新演算並與所接收到的 FCS 比對，如果不相符，表示有 CRC 錯誤。CAN bus 不同於一般的 CRC 演算法，CAN bus 的 FCS 共 15 位，演算法如下：

   $x^{15}+ x^{14}+ x^{10}+ x^{8}+ x^{7}+ x^{4}+ x^{3}+ x^{0}$

2. Frame check

   檢查封包中幾個固定值的欄位以驗證該封包是否有被訊號干擾導致內容錯誤。

3. ACK errors

   接收端在收到封包後會告知發訊端，發訊端若沒有收到確認訊息，ACK 錯誤便發生。接收到影格而沒有發現錯誤的每個節點在 ACK 欄位中傳送顯性訊號，來覆蓋發射器的隱性訊號。

4. Monitoring

   傳輸一位到網路上，再從網路讀取來檢查是否一致。

5. Bit stuffing

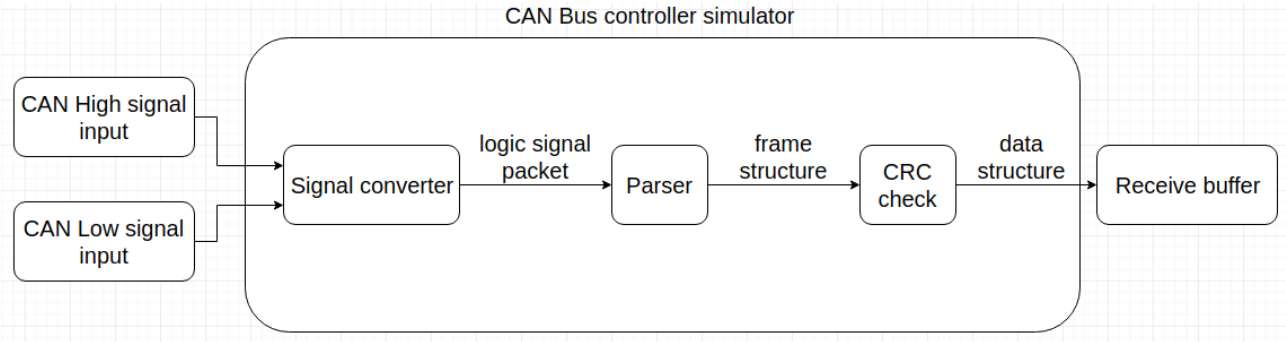
用於訊息同步，傳輸器會在相同極性的五個連續位之後插入一個相反的極性的位，以確保足夠的轉換來保持同步。

6. 這次作業中有實做的部份為 CRC、Frame check、Bit stuffing(Bit stuffing 只有實做一半，在下面說明)。

Introduction of the parser

由於 CAN bus protocol 一般都是由硬體實做，包括電線訊號轉邏輯訊號、仲裁機制、parse packet、error checking 之類的，所以這次作業我們寫了一個 CAN bus controller simulator，我們去模擬硬體的 signal convert, parse packet, error checking 的機制。

1. Architecture



1. 最左邊的兩個輸入分別是一開始寫好的檔案，用來模擬電線訊號的輸入。(High.txt and Low.txt)，內容是好幾個 0, 1, -1。0 代表穩態，1 代表 CAN Hi 被驅動，-1 代表 CAN Lo 被驅動。
2. 主要分成 3 個 components，第一個 component 主要的工作是把電線訊號轉成邏輯訊號，第二個 component 主要的工作是把邏輯訊號轉成 frame structure，第三個 component 負責做 error checking。
3. 以實際硬體來說最後 CAN bus controller 會把資料存入 register 並觸發 interrupt 來讓系統處理，這邊我們把結果輸出到螢幕來模擬存入暫存器。

2. Function Call Graph

main.c

function	call function
main()	receive()

signal_converter.c

function	call function
receive()	signal_convert() bit_stuffing_check() frame_check() parse() crc_check() save_data()
signal_convert()	
save_data()	

parser.c

function	call function
parse()	

error_checking.c

function	call function
bit_stuffing_check()	
frame_check()	
crc_check()	

3. Function Prototype and detail

typedefine

- **logic_signal_packet**
  - 因為一個 packet 加上 bit stuffing 不會超過 160 bits 但可能超過 128 bits，所以用 160 bits 去存。
  - 在轉換時順便紀錄 packet_len。

```
typedef struct
{
    uint32_t bits[5];
    uint32_t packet_len;
} logic_signal_packet;
//the first bit is msb
```

- **frame_struct**

```
typedef struct
{
    uint32_t ID_A;
    char SRR:1;
    char IDE:1;
    uint32_t ID_B;
    char RTR:1;
    char Res:2;
    char DLC:4;
    char data[9];    //the last element is for '\0'
    uint32_t CRC:15;
    char CRC_del:1;
    char ACK:1;
    char ACK_del:1;
    char end_of_frame:7;
} frame_struct;
```

main.c

- int main()
  - 直接呼叫 receive()
  - receive return 後就結束程式

signal_converter.c

- void receive()
  - 讀檔(High.txt and Low.txt)，內容是好幾個 0, 1, -1
  - 每讀 1 bit 就 call 一次 signal_convert(int h, int l)，signal_covert 回傳 logic signal
  - 將 logic signal 存進 logic_signal_packet
  - 當一個 packet 完成後 call bit_stuffing_check(logic_signal_packet*)，若失敗則 print 錯誤

訊息並繼續讀下一個 packet(return 值 0 為成功，-1 失敗，後面同)，否則繼續往後做。
    - call frame_check(logic_signal_packet*)
    - call parse(logic_signal_packet*, frame_struct*)
    - call crc_check(frame_struct*)
    - save_data(frame_struct*)
      - print the ID and data content
      - printf("ID:%d, data:%s, packet count:%d, packet length: %d\n", frame.ID_A, frame.data, count, packet_length);
- int signal_convert(int h, int l)
    - convert can bus signal to logic signal
- void save_data(frame_struct*)
    - print the ID and data content
    - printf("ID:%d, data:%s, packet count:%d, packet length: %d\n", frame.ID_A, frame.data, count, packet_length);

parser.c

- void parse(logic_signal_packet*, frame_struct*)

error_checking.c

- int bit_stuffing_check(logic_signal_packet*)
    - error checking
    - return 值 0 為成功，-1 失敗
    - bit stuffing 的測資部份我們只有連續五個 0 會補 1，連續五個 1 並沒有補 0。原因是我最後做 report 時才發現是不管連續五個 0 或 1 都要補一個反向位元(不小心沒注意看清楚協定內容)，而我在定 spec 的時候其他同學都依照我的錯誤觀念完成自己的部份了，我們來不及修改程式碼及測資。但 bit stuffing 邏輯部份是正確的。
- int frame_check(logic_signal_packet*)
    - error checking
    - return 值 0 為成功，-1 失敗
- int crc_check(frame_struct*)
    - error checking
    - return 值 0 為成功，-1 失敗

4. **The way to execute our parser**
    1. $ make
    2. $ make parse
    3. $ make compare

The contribution of each member

- 李俊德
    - Make the spec
    - Integration test
    - Fix bug
    - Write the report
- 蕭丞志
    - The component "signal converter"
    - Unit test
    - Fix bug
- 王昱翔

    - The component “parser”
    - Unit test
    - Fix bug
- 蘇致翰
    - The component “error checking”
    - Unit test
    - Fix bug

Reference

- ncku csie wiki
- wkikipedia
- CRC-15