University of British Columbia
Electrical and Computer Engineering
CPEN 412 - Microcomputer Systems Design

Assignment 6B

Objective

- Add a Canbus Controller to our 68k computer system, write the driver software and network 2 or more Canbus nodes together to exchange and display messages with each other.
- This is mainly a software assignment to drive the Canbus controller and develop a simple "messaging" application.

Rational

Networking microcomputer systems has become much more common in industrial applications as it allows computers to be distributed and placed next to equipment that they are controlling. They can then communicate with other microcomputers perhaps hundred of meters away.

A particularly common real-time, dependable network is based around **Canbus**. The purpose of this lab then is gain some experience with CanBus concepts and network two "nodes" together, sending messages from one node across the network to another.

Part A – The Hardware - Step 1

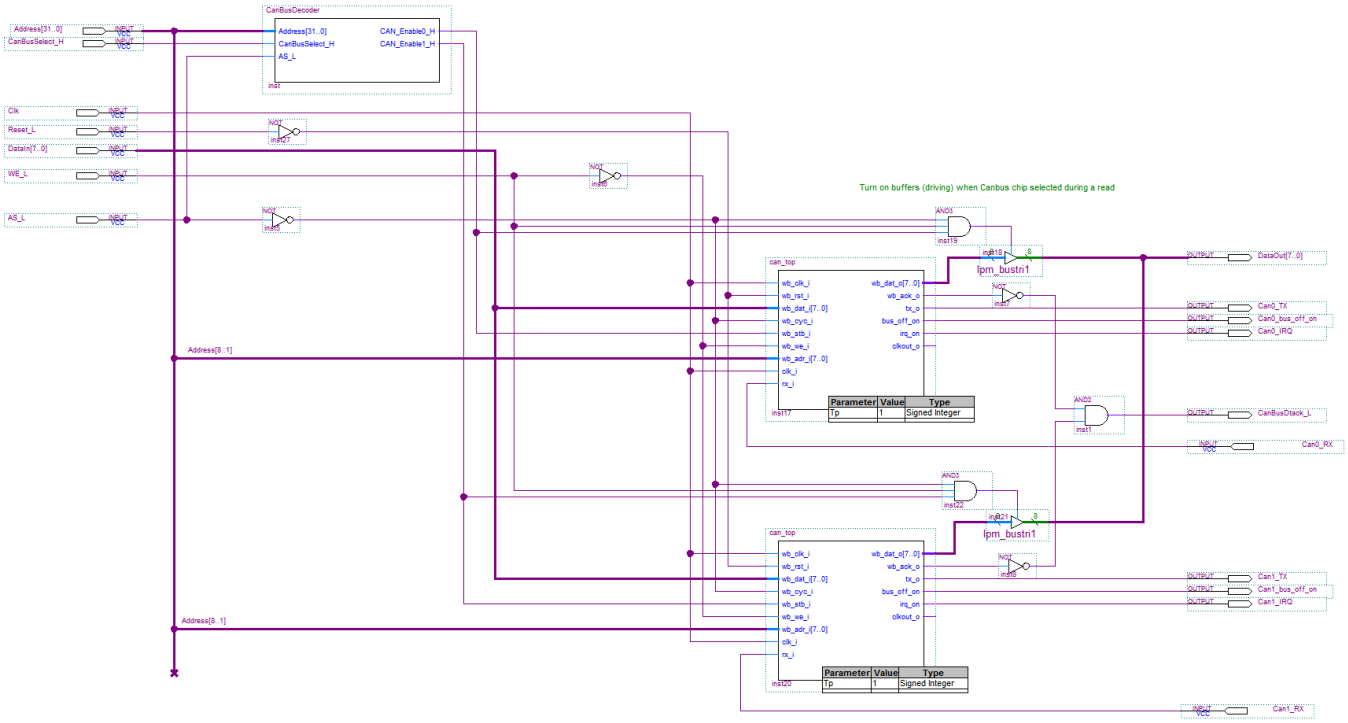
On Canvas you will find a zipped folder with 13 Verilog files (downloaded from **OpenCores.org**) that define an implementation of one of the first industry standard Canbus controllers, the SJA 1000 made by Philips Semiconductors (*now called NXP*). You will also find a data sheet for that controller.

You can buy this same controller today as a stand alone chip for about $10. Check out digikey here http://www.digikey.ca/product-detail/en/nxp-semiconductors/SJA1000T-N1,118/568-1123-1-ND/735806

Download and unzip the folder and add the 13 verilog files to your project (*open them one at a time and tick the "add to project" check box for each*). The file "can_top.v" represents the top level of the design hierarchy for this chip, so create a **symbol** for that file so that we can paste it onto our 68k schematic diagram later.

Step 2

Create a new schematic/block diagram file in Quartus and paste 2 copies of the can_top symbol onto it and wire up as shown in the schematic below. Having 2 Can controllers allows us to network with *ourselves*, on the same board, to test and debug the software and hardware. Once this is working, we could network two Altera boards, in which case we will only really need 1 controller on each board **but as we are working remotely, we will limit our assignment to networking between two nodes on the same DE1SoC using two controllers as shown below.**



You will notice that this device has a **Wishbone** compliant bus interface, like the I^2C controller we used in the previous assignment, and is able to produce its own **Dtack** signal, which we can feed back to the 68k. Both device Dtack signals have been combined into a single active low **CanBusDtack_L** signal using an **AND** gate.

An outline of the VHDL file for the **Canbus (Address) decoder** at the top of the above schematic is given below. Modify this to suit the range of addresses for the two controllers as detailed in the **TODO** comments below. The addresses chosen for the controllers are fairly arbitratry, and were chosen so that they do not clash with other devices or memory in the memory map of our system.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity CanBusDecoder is
```

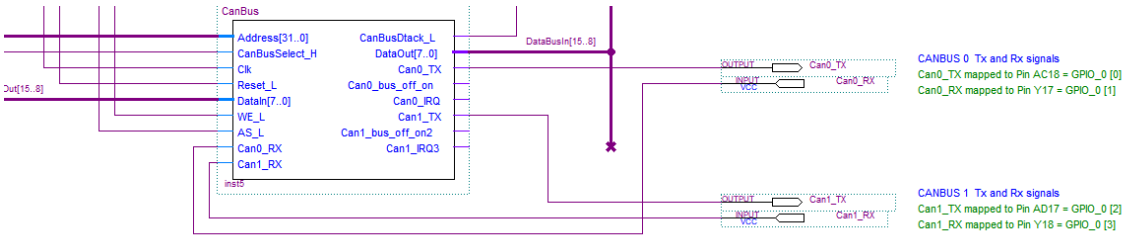
```
        Port (
                Address                 : in Std_logic_vector(31 downto 0) ;
                CanBusSelect_H          : in Std_logic ;                        -- active when 68k accesses range 0x00500000-0050FFFF
                AS_L                    : in Std_Logic ;

                CAN_Enable0_H           : out Std_logic ;
                CAN_Enable1_H           : out Std_logic
        );
end ;

architecture bhvr of CanBusDecoder is
Begin
        process(Address, CanBusSelect_H, AS_L)
        Begin
                CAN_Enable0_H  <= '0' ;             -- default is NOT active
                CAN_Enable1_H  <= '0' ;

-- Design for the Canbus Controller decoderrs. Each controller occupies 256 bytes
-- of memory space. This is why CAN controller has (A7..0) pins, but chip has an 8 bit
-- data bus and is mapped to upper half of 68k's data bus, so address should be connected to
-- (A8..1) thus each device occupies 512 bytes in the 68k's space.
--
-- Using CanBusSelect_H signal above to reduce decoder expression/complexity, design the following

-- TODO : enable Can0 if 68k is accessing an address in range $00500000 - $005001FF
-- TODO : enable Can1 if 68k is accessing an address in range $00500200 - $005003FF

        end process;
END ;
```

Step 3

Create a new symbol for the Can Controller schematic above and paste it on to the top level 68k schematic and wire as shown below. Some of these signals will be connected to the main 68k address decoder and Dtack generator (*see later*).

Connect the clock signal (**Clk**) to the **25Mhz clock** used by the 68k and the **DataIn** and **DataOut** signals to the upper half of the 68k's data bus (signals **8..15**). You can ignore the **CanX_bus_off_on** signals.

Bring out the **RX** and **TX** signals for each Canbus Controller onto IO pins as shown below. Once you have finished the whole design and compiled it **remember to go back and make the connections on the PIN planner for these TX/RX signals as shown below for a DE1 board (they may already exist in the project) and recompile.**



This will bring the signals out to the GPIO_O connections shown below

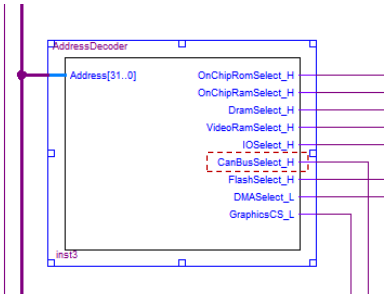
JP1 (GPIO 0) Connector (LHS) - [x] = GPIO_O[x]

Can0_TX mapped to Pin AC18 = GPIO_0 [0] Can1_TX mapped to Pin AD17 = GPIO_0 [2]	1 3	[0] - AC18 [2] - AD17	[1] - Y17 [3] - Y18	2 4	Can0_RX mapped to Pin Y17 = GPIO_0 [1] Can1_RX mapped to Pin Y18 = GPIO_0 [3]
	5	[4] - AK16	[5] - AK18	6	
	7	[6] - AK19	[7] - AJ19	8	
SCL mapped to Pin AJ17 = GPIO_0 [8]	9	[8] - AJ17	[9] - AJ16	10	SDA mapped to Pin AJ16 = GPIO_0 [9]
Vcc 5v	11			12	GND
	13	[10] - AH18	[11] - AH17	14	
	15	[12] - AG16	[13] - AE16	16	
	17	[14] - AF16	[15] - AG17	18	
DataBusIn[0]	19	[16] - AA18	[17] - AA19	20	DataBusIn[15]
DataBusIn[1]	21	[18] - AE17	[19] - AC20	22	DataBusIn[14]
DataBusIn[2]	23	[20] - AH19	[21] - AJ20	24	DataBusIn[13]
DataBusIn[3]	25	[22] - AH20	[23] - AK21	26	DataBusIn[12]
DataBusIn[4]	27	[24] - AD19	[25] - AD20	28	DataBusIn[11]
Vcc 3.3v	29			30	GND
SCK_O	31	[26] - AE18	[27] - AE19	32	SSN_O[0]
MOSI_O	33	[28] - AF20	[29] - AF21	34	MISO_I
DataBusIn[5]	35	[30] - AF19	[31] - AG21	36	DataBusIn[10]
DataBusIn[6]	37	[32] - AF18	[33] - AG20	38	DataBusIn[9]
DataBusIn[7]	39	[34] - AG18	[35] - AJ21	40	DataBusIn[8]

Pin Numbers 1 - 40

Step 4

Modify the main address decoder on the top level 68k schematic diagram to set aside a 64Kbyte block of address space for our 2 Can controllers. It should generate a **CanBusSelect_H** signal (*needed by the secondary decoder on our CanBus circuit above*) for addresses in the range **hex [0050 0000 - 0050 FFFF]**. You can see this signal in the image below:-

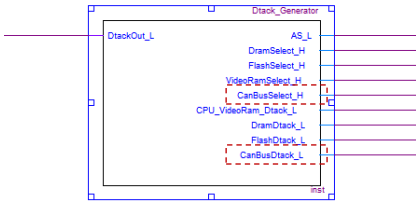


Step 5

Modify the main Dtack generator circuit on the top level 68k schematic so that whenever the **CanBusSelect_H** signal above is driven to '**1**' (that is the 68k is accessing an address in the range [**0050 0000 - 0050 FFFF]**), the Dtack presented to the 68k will come from the CanController dtack e.g.

```
elsif(CanBusSelect_H = '1') then
        DtackOut_L <= CanBusDtack_L;        -- wait for Canbus dtack
end if ;
```

You can see these signals integrated into the Dtack generator circuit below

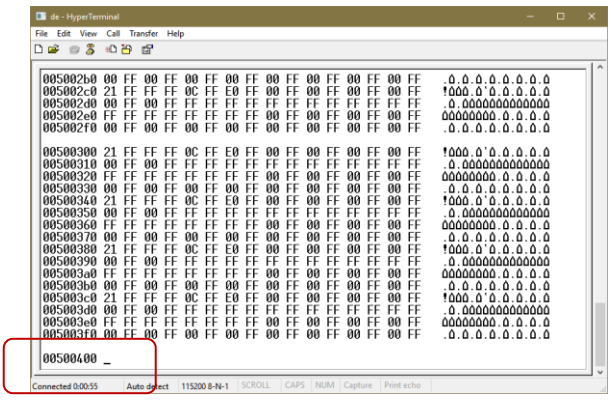


Step 6 - Compile the design, check that you have made the connections for the **TX/RX** signals for each Can controller in the Pin planner, then download the "**.sof**" file to your Altera board.

If you have done the above steps correctly, you should be able to use the debug monitor to verify that you can communicate with the 2 Can controllers by **dumping** the memory space occupied by the controllers. You should see something like this e.g. **hex 21** in the first location and then the pattern is replicated multiple times – this is is a feature of the design of the Can controller.

The image below shows the **2nd Can controller address space** being dumped which occupies the address range [0050 0200 – 0050 03FF].

Note how the debug monitor "**hangs**" when it tries to dump memory starting at address **0050 0400**. This is to be expected. Can you explain why this happens?



If you can see this pattern of data, then the hardware at least is working.

Step 7
You will be given **two 8 pin Canbus Transceiver devices** in a DIP package that you can place onto a breadboard. This device converts ground referenced digital signal data into a differential (2 wire) signal **CANH** and **CANL** for transmission of data and vice versa for reception of data.

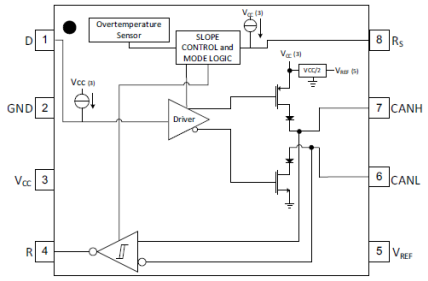
The chips we have are actually made by Microchip (Part # **MCP2551** – see data sheet on Canvas). An example image of the chip is shown below in surface mount form. Your's uses a breadboard friendly DIP packaging. Make sure you use the correct chips as your instructor has supplied several in 8 pin DIP packaging. Use a cell phone camera to zoom in on the package to read the part # if required.

The pin out for the signals on the transceiver is given below. The device can be powered from the **5v/0v** signals on the GPIO connector.

Make sure you can indentify Pin 1 on the device. NOTE that **ground** and **Vcc** are on **pins 2** and **3** – which is very unusual. If you connect power the wrong way around you **<u>will</u>** literally blow up the chips. They will smoke and give off a nasty smell so be VERY careful (you may not be able to get replacements in time for the assignment deadline.). **Triple** check the connections before applying power.

V_{ref} can be ignored, while R_s can be connected to Ground (**0v**). The signals **CANH** and **CANL** are the differential network signals that we use to connect to the network and join nodes together. The other pins are described below. D is the data input to the device (which connect to the *transmit data* output of our canbus controller), and R is the received data output (which connects to the *receive data* input of our canbus controller)

Block Diagram



Pin Functions

PIN		I/O	DESCRIPTION
NAME	NO.		
CANH	7	I/O	High-level CAN bus line
CANL	6	I/O	Low-level CAN bus line
D	1	I	CAN transmit data input (LOW for dominant and HIGH for recessive bus states), also called TXD, driver input
GND	2	GND	Ground connection
R	4	O	CAN receive data output (LOW for dominant and HIGH for recessive bus states), also called RXD, receiver output
R_S	8	I	Mode select pin: strong pulldown to GND = high-speed mode, strong pull up to V_{CC} = low-power mode, 10-kΩ to 100-kΩ pulldown to GND = slope control mode
V_{CC}	3	Supply	Transceiver 5-V supply voltage
V_{REF}	5	O	Reference output voltage

Connect a transceiver chip to each of your two Can controllers by making connections from the appropriate pins on the **GPIO 0** connector on the Altera board, to the transcievers on your breadboard.

- The **D** pin above is the Data **input to** the Can Transceiver (from the Can controller) and connects to your **Can0_TX** (or **Can1_TX**) pin on the **GPIO_O** connector.
- The **R** pin above is the Data **output from** the Can Transceiver and connects to the **Can0_RX** (or **Can1_RX**) pin on the **GPIO_O** connector.

Now connect the two transceiver chips togther with a pair of short wires (**CANH** to **CANH** and **CANL** to **CANL**) to form a 2 wire differential network with 2 Can nodes, (*see example in Figure 30 in the Texas Instruments transceiver data sheet posted on Canvas*). There is **no** need to **terminate** the network with **120 ohm** resistors, since the network is short and we'll only run it at **100kpbs**, but a longer faster network should be terminated to avoid signal reflections at each end.

Part B – Writing the Software

On Canvas you'll also find an **Application note** for the SJA1000 Canbus Controller. This document describes the operation of the SJA1000 chip (*which our soft core verilog design implements*). There's also a data sheet for the device itself. Both data sheets can seem daunting when you first open them as the chip has **many registers**, but the

application note does a reasonable job of explaining how the chip works and what each register (**and each bit within each register**) does and also how to initialise the device and how to transmit and receive a CAN message.

Of course we won't be implementing all the features of this chip e.g we'll **ignore** error checking and error counters etc. and just focus on transmitting and receiving simple 8 byte (or smaller) messages. As is the way with embedded hardware, we focus on writing basic routines to "get something going" and then, when time and experience permit, we write more sophisticated routines.

Note that the device can be programmed to work in one of two modes. **Basic Can** and **PeliCan** (*a more versatile mode*). We will be using **PeliCan mode** which has to be programmed to select it during **initialisation**. In addition, note that some of the registers in the chip are **read only**, some **write only**. Some registers are only accessible while in **reset mode** and some registers are only accessible while in **operating mode**. The transition from Reset to operating mode also has to be made by your 'C' code after initialisation.

The read/write operation of the chip maintains cyclic buffers for multiple incoming/outgoing messages (up to 11 bytes in total per message including the overheads of the ID etc) which explains why each Can Controller chip occupies 256 bytes of the 68k's address space.

The speed of the network (bits per second) is programmable. To keep it simple, we will be using timing derived for **100kbits per second** assuming a **25Mhz clock**. The timings were derived with the help of this tool https://www.kvaser.com/support/calculators/bit-timing-calculator/

(*the timings have been done and are included in the example code*).

There are also examples of C code routines to initialise the device and read/write messages from/to it in the application note and also template code on Canvas. Note also that some code in the application sheet is related to programming an **8051** used as the example CPU in the application note, so that obviously won't need to be included. In addition, to keep it simple, you should use **polling** rather than **interrupt** driven IO.

Your task here is to implement a simple messaging application between the two nodes on a single Altera board. If you are able, you might like to partner up with another student and network between two or more different DE1 boards.

What do I have to do?

1. **Using Timer Interrupts**

   Assign one of your Canbus nodes it's own **hard coded message numbers** (via software) and then send messages from that node to the other node. Four type of messages should be broadcast

   a. The status of the lower **8 slider switches** (as a single byte), **every 100mS**.
   b. The value of the **ADC** potentiometer(from Lab 5) every **200ms**.
   c. The value of the **Light sensor** every **500ms** and
   d. The value of the **Thermistor** (temp setting) every **2 secs**.

   You should use an interrupt and one of the spare timers on the 68k system (there are 8 and we used one in Assignment 6A for the uC OS/II Operating system). Trace the timers IRQ outputs to see which 68k interrupt level they are using. They should all be wired to Level 3 IRQ through a big 8 input gate).

   Modify the hardware to wire one of the timers to a **Level 6 IRQ** and use that. Revisit Lecture 17 - IO Interfacing and Interrupts.pptx to refresh your memory of how exceptions work on the 68k

   For details of how to use interrupts and install interrupt handlers, take a look at the C code in the **M68kUserProgram (DE1).c** file that should be included as part of your Quartus Project (See folder **Programs->Debug Monitor Code**).

   The function **InstallExceptionHandler()**, allows you to intercept a 68k IRQ and route it to one of your own C code functions to deal with. Also make sure the **StartOfExceptionVectorTable** constant define at the top is correct for a Dram based system.

   The **switch** and three **ADC** values should then be printed out on the screen (hyperterminal window) after being read by the receiver node (as part of the main() program code **80%**

2. Once you have done that and verified that you can network two nodes, break down the application to make use of **threads** (as per the uC/OS II kernel we ported in Lab 6A).

   Four threads (using sleep based time delays) could read the switches plus the three ADC values and send the messages via one canbus node, a fifth thread, could be reading the 2nd canbus node and displaying the data sent. Depending upon how you do this, you may need a mutex to protect the Canbus Controller

(uC/OS II supports Mutexes) transmitting the data from simultaneous access by several threads **20%**

To get you up to speed, it will obviously help a lot to read the lecture notes on CanBus that were given in class so that you understand the basic concepts. You should also read and attempt to understand the SJA1000 application note (*at least as much as you can before starting*).

A 'C' file has also been posted on Canvas with the basic **#defines** and **register/bit** definitions contained within, that you can use as part of your assignment. There are also "**place hangers**" for you to write your initialisation/read/write routines.

Submission
Zip the following and submit to Canvas. **Only one student in your group (if you worked with a partner) needs to do this**

- **The "Lab Student Contribution Sheet" from Canvas for this Lab 6 (complete 1 sheet to cover both 6A and 6B)**
- **All C code for the "multi-threaded" problem from 6A**
- **All C code for the Interrupt driven version of the messaging application**
- **All C code for the Multi-threads, uC OS/II version of the messaging application**

Furthermore, provide in your submission comments in Canvas a YouTube link to a short video (where the video includes narration on what is happening) that has you demonstrating

1. **Shows the multi-threading problem from 6A running**
2. **Shows the Canbus transceiver chips plugged into the Breadboard and wired to the DE1 board.**
3. **Demonstrates the messaging application (interrupt and multi-threading). For this, include video that shows the switches and adc values changing and the messages being sent.**

   Show also that if you "pull" the network wires, the application stops and when your reconnect, the messaging resumes.