**Description and specifications**
For this project we will receive and transmit messages over the Control Area Network (CAN) bus. In EE 354/454 we used the Atmel AT89C51AC3 processor for several projects. This same processor is available with a fully implemented onboard CAN controller under the name AT89C51CC03. The CAN bus uses just two wires for signals and a third wire for a ground. The AT89C51CC03 has the two CAN bus data signals on port 4 at P4.0 and P4.1. The CAN bus requires a bus driver and we will use the ATA6660 from Atmel for this purpose. We will transmit and receive signals over the CAN bus.

**Atmel AT89C51CC03**
The AT89C51CC03 is an Atmel processor with an 8051 core which implements the full CAN bus in versions 2.0A or 2.0B. The AT89C51CC03 is identical to the Atmel AT89C51AC3 (which we used in EE 354/454) with a full CAN controller added. Two pins which are marked P4.1 and P4.1 on the AT89C51AC3 are used as the CAN data lines CANH and CANL.

- 64 K on chip flash memory with 2K flash serial bootloader and 2K bytes of EEPROM.
- Three 16-bit timers/counters
- A 20MHz crystal that is double clocked. (This can go to 30MHz.)
- 32 I/O lines P0, P1, P2, P3, and 2 bits of P4
- Five channels of 16-bit PCA with 8-bit PWM
- Double data pointer
- 8 channel x 10-bit A to D converter on P1
- A 10-bit A/D converter with 8 multiplexed analog inputs
- Full CAN controller for 2.0A and 2.0B

![Figure 5](image-url)
Block diagram of the Atmel 89C51CC03 microcontroller.
The AT89C51CC03 CAN Controller

The AT89C51CC03 CAN controller has the following characteristics:

- Fully Compliant with CAN Rev 2.0A and 2.0B
- 15 Independent Message Objects (programmable for transmission or reception)
- 8-byte Cyclic Data Register (FIFO)/Message Object
- 16-bit Status and Control Register/Message Object
- 16-bit Time-Stamping Register/Message Object
- Access to Message Object Control and Data Registers Via SFR
- Supports
  - Time Triggered Communication
  - Autobaud and Listening Mode
  - Programmable Automatic Reply Mode

A block diagram of the on-board CAN controller is shown in Figure 6.

Figure 6
The AT89C51CC03 CAN controller.

The AT89C51CC03 CAN controller has five sources of interrupts which are:

- Transmission interrupt,
- Reception interrupt,
- Interrupt on error (bit error, stuff error, crc error, form error, acknowledge error),
- Interrupt when Buffer receive is full,
- Interrupt on overrun of CAN Timer.

The controller also has the ability to be operated in a polling mode. For this class we will largely ignore CAN errors and interrupts and learn to transmit and receive CAN messages in a polled program.

CAN SFRs
There are 34 CAN Special Function Registers. These are summarize in Figure 7 along with their function.
### Table: Mnemonic and Name and Description

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANGCON</td>
<td>CAN General Control - 8 control bits that effect the CAN controller</td>
</tr>
<tr>
<td>CANGSTA</td>
<td>CAN General Status - 8 status bits for the CAN controller</td>
</tr>
<tr>
<td>CANGIT</td>
<td>CAN General Interrupt - holds the 7 interrupt bits. Can be polled</td>
</tr>
<tr>
<td>CANBT1, CANBT2, CANBT3</td>
<td>CAN Bit Timing 1. These three registers hold the baud rate data and the bit timing data for clock synchronization.</td>
</tr>
<tr>
<td>CANEN1, CANEN2</td>
<td>CAN Enable Channel byte 1 and byte 2. Bits in these registers enable the 15 message channels.</td>
</tr>
<tr>
<td>CANGIE</td>
<td>CAN General Interrupt Enable. Holds bits to enable the 7 CAN interrupts</td>
</tr>
<tr>
<td>CANIE1, CANIE2</td>
<td>CAN Interrupt Enable Channel byte 1, 2. These bits enable interrupts for the 15 messages.</td>
</tr>
<tr>
<td>CANSIT1, CANSIT2</td>
<td>CAN Status Interrupt Channel byte1. Holds bits that tell if an interrupt is turned on for each of the 15 channels.</td>
</tr>
<tr>
<td>CANTCON</td>
<td>CAN Timer Control. 8-bit prescaler for the main CAN timer</td>
</tr>
<tr>
<td>CANTIMH, CANTIML</td>
<td>CAN Timer high, low. The main CAN timer</td>
</tr>
<tr>
<td>CANSTMP H, L</td>
<td>CAN Timer Stamp high, low. 16 bit time stamp for messages</td>
</tr>
<tr>
<td>CANTTCH, L</td>
<td>CAN Timer TTC high, low. Time Triggered Communications. This protocol allows the CAN network to send messages based on events at periodic intervals.</td>
</tr>
<tr>
<td>CANTEC, REC</td>
<td>CAN Transmit/Receive Error Counter.</td>
</tr>
<tr>
<td>CANPAGE</td>
<td>CAN Page. Each message is on a page in memory. CANPAGE is the index and is typically auto-incremented</td>
</tr>
<tr>
<td>CANSTCH</td>
<td>CAN Status Channel. The status for each channel</td>
</tr>
<tr>
<td>CANCONCH</td>
<td>CAN Control Channel. The Control register for each channel</td>
</tr>
<tr>
<td>CANMSG</td>
<td>CAN Message Data. Typically this is an 8-byte array</td>
</tr>
<tr>
<td>CANIDT1, 2, 3, 4</td>
<td>CAN Identifier Tag byte 1(Part A) CAN Identifier Tag byte 1(Part B) This is the ID data for the message.</td>
</tr>
<tr>
<td>CANIDM1, 2, 3, 4</td>
<td>CAN Identifier Mask byte 1(Part A) CAN Identifier Mask byte 1(Part B). This is a mask for the ID that allows you to received multiple messages with one ID.</td>
</tr>
</tbody>
</table>

**Figure 7**
The 34 CAN Special Function Registers.

**Setting up the CAN controller to receive and transmit messages**

To successfully write a program which can receive and transmit data over a CAN network we will need to set up all of the registers in Figure 7 except for those which are shaded since we are not using interrupts and we will ignore error messages.

To begin we need to initialize the CAN General Control register CANGCON, disable the interrupts, set up the bit timing registers CANBT1, 2, and 3, and for each of the 15 possible message objects we will initialize the CAN Control Channel registers (CANCONCH), the CAN Status registers for each channel (CANSTCH), and the ID and mask registers.

The CANGCON register is shown in Figure 8. For initialization we do a general reset and disable the CAN controller.

\[\text{CANGCON} = 0x01.\]
As part of the initialization process we also reset all of the mailboxes and the CAN message data register for each channel.

```c
// Reset all mailboxes, IDs, masks and message data registers
for (num_channel = 0; num_channel < 15; num_channel++)
{
    CANPAGE = num_channel << 4;
    //CANCONCH = CAN Message Object Control and DLC Register
    CANCONCH = CH_DISABLE; //Disables transmission and reception messages
    CANSTCH = 0; //CAN Message Object Status Register
    CANIDT1 = 0; //CAN Identifier Tag Register 1
    CANIDT2 = 0; //Set all of the identifier tags to 0
    CANIDT3 = 0;
    CANIDT4 = 0;
    CANIDM1 = 0; //CAN Identifier Mask Registers 1
    CANIDM2 = 0; //0 masks out ID bits so ID disabled
    CANIDM3 = 0;
    CANIDM4 = 0;
    for (num_data = 0; num_data < 8; num_data++)
        CANMSG = 0; //CAN Message Data Register. MSG register address is
                      // auto incremented so this makes all 8 messages = 0
}
```

There are two interrupts that need to be disabled for initialization. One is for the CAN controller and the other is for the timer. Set both of these enable bits to 0. ECAN and ETIM are both defined bits in the .h file.

```c
ECAN = 0;
ETIM = 0;
```

The bit timing registers are tedious to set up. Figure 9 shows the parameters involved in bit timing.
The relevant parameters that are needed are:

- **BRP** – Baud Rate Prescaler. This reduces the system clock to the right bit rate for the baud rate chosen.
- **PRS** – PRopagation time Segment.
- **PHS1** – Phase Segment 1
- **PHS2** – Phase Segment 2
- **SJW** – Synchronization Jump Width.

In Figure 9, the time period marked by \( T_{sq1} \) is referred to as one time quanta or TQ. PRS and PHS1 must be 1 to 8 TQ long. Ideally we would like the sample point to be about 75% of the bit time. Fortunately, these calculations are tedious enough that Atmel (and other companies) provide a Java program\(^1\) that can be used to do the calculations and get the values to place into the relevant CANBT registers.

Using the Atmel Bit Configurator as shown in Figure 10 we see that we can have a sample point at 75%, a 12MHz clock, and 500 KBaud with BRP = 0 (no Baud rate prescaling), \( t_{PRS} = 2 \), \( t_{PHS1} = 6 \), and \( t_{PHS2} = 3 \). (If this was in x2 mode we would set the system frequency to 24MHz.) In general we make the Synchronization Jump Width as small as we can. Setting it to 0 will cause it to default to 1 TQ. This selection give **CANBT1** = 0, **CANBT2** = 0x02, and **CANBT3** = 0x2B.

---

\(^1\)Go to: http://www.atmel.com/dyn/products/product_card.asp?part_id=3186 and download the "CAN bitrate configurator for 8051 and AVR microcontrollers". This is a JAVA applet.
In a program we define the following constants to make them easily changeable:

```c
#define BT1 0x00
#define BT2 0x02
#define BT3 0x2B
```

In the program we can set up the CANBT registers like this.

```c
CANBT1 = BT1;  //BT1 is the baud rate generator. BRP=0x00;
CANBT2 = BT2;  //BT2 has tSJW and tPRS
CANBT3 = BT3;  //BT3 has tPHS1 and tPHS2
```

At this point we can start the CAN controller running.

```c
CANGCON |= MSK_CANGCON_ENA; //Start the CAN Controller state machine.
```

Finally, we need to set up the control register, status register, ID, and mask for the 15 channels. Since we are only going to transmit and receive we only need 2 channels so we will neglect the other 13. We will use Channel 0 for reception and Channel 1 for transmission. To address the registers for a channel we must first set the memory page where the registers are stored. We do this by setting a value in CANPAGE shown in Figure 11.
CAN Message Object Page Register

<table>
<thead>
<tr>
<th></th>
<th>CHNB 3</th>
<th>CHNB 2</th>
<th>CHNB 1</th>
<th>CHNB 0</th>
<th>AINC</th>
<th>INDEX2</th>
<th>INDEX1</th>
<th>INDEX0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit channel number goes to bits 4-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>auto-incr</td>
<td>0 = on</td>
<td>Channel register</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11**

CANPAGE register. The top four bits specify the channel. The lower three bits specify the register within the channel. The AINC bit specifies whether or not the index is auto-incremented.

CANPAGE = (0 << 4); // CHNB=0x00; select channel 0. Bits 4 and 5 = 00
CANSTCH = 0x00; // reset channel status
CANCONCH = CH_DISABLE; // reset control and dlc register

The CAN ID numbers are 11 bits for version A and an additional 19 bits for version B. We are using version A only. The 11 bits are stored in CANIDT1 and CANIDT2 in the 11 least significant digits. To set the ID to say 0x123 = 0000 0001 0010 0011 we do this:

/* Channel 0: identifier = 11bits. CANIDT=0x123 = 001 0010 0011 */
CANIDT1 = 0x24; //CANIDT1 is bits 3 to 10 of the ID = 0010 0100
CANIDT2 &= ~0x80; //CANIDT2 has bits 0 - 2 in positions 5, 6, and 7 = 011
CANIDT2 |= 0x60;

This ID number means that we are only going to look at messages which have an ID = 0x123 and ignore all others. We can however, further modify this ID field with the ID mask. The ID masks allows us to mask out (with zeros) the ID field of messages to allow for a wider range rather than just one ID number. The mask bits are stored in CANIDM1 and CANIDM2. Here's an example of setting the mask bits.

/* Channel 0: mask = 11bits. 0x7F0 = 111 1111 0000*/
CANIDM1 = 0xFE; //CANIDM1 has bits 3 to 10 = 1111 1110
CANIDM2 &= ~0xE0; //CANIDM2 has bits 0-2 in positions 5, 6, and 7 = 000
CANIDM4 = 0; //Disables remote transmission & makes 0 higher priority than 1
CANIDT4 &=~0x04; //Set Remote Transmission Request to 0 in IDT4

Finally we need to set up the data length in bytes and enable reception for this channel. For the CC03 the maximum data length is 8 bytes. The CANCONCH register is shown in Figure 12.

CANCONCH |= DLC_MAX; //DLC_MAX = 8. Bits 0-3 set maximum data length bytes
CANCONCH |= CH_RxENA; //CH_RxENA = 0x80. Bits 6 & 7 enable reception w/o buffer

This mostly completes the setup for Channel 0 as a receiver. We next need to set up Channel 1 as a transmitter. The process is similar. We begin by setting up the page number for Channel 1 along with the channel status and control register.
July 21, 2009

**CAN Message Object Control and DLC Register**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONCH 1</strong></td>
<td><strong>CONCH 0</strong></td>
<td><strong>RPLV</strong></td>
<td><strong>IDE</strong></td>
<td><strong>DLC 3</strong></td>
<td><strong>DLC 2</strong></td>
<td><strong>DLC 1</strong></td>
<td><strong>DLC 0</strong></td>
</tr>
<tr>
<td>0 0 Disable</td>
<td>Reply</td>
<td>0 = RevA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 Launch Xmit</td>
<td>valid</td>
<td>1 = RevB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 enable recpt</td>
<td>after</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 enable recpt</td>
<td>remote</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with buffer</td>
<td>frame</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 12**
The CAN Control register for the Channel.

CANPAGE = (1 << 4);   //Bits 4-5 in CANPAGE = 1 for Channel 1
CANSTCH = 0x00;       //Reset the channel status
CANCONCH = CH_DISABLE; // reset control and dlc register

The Channel 1 identifier will be an 11 bit number stored in CANIDT1 and CANIDT2. We will set this to 000 0000 0001.

```c
/* Channel 1: identifier = 11bits. CANIDT=0x001 = 000 0000 0001*/
CANIDT1 = 0x00;       //CANIDT1 is bits 3 to 10 of the ID = 0000 0000
CANIDT2 &= ~0xC0;     //CANIDT2 has bits 0-2 in positions 5, 6, and 7 = 001

Since Channel 1 is used for transmission it is not necessary to set up a mask register.

**A Main program for transmission and reception**

For a main program we will set up a button on external interrupt 0 and we will send a message over the CAN network each time it is pushed. We will otherwise, poll the interrupt line for Channel 0 reception to collect CAN messages sent to Channel 0. We will retransmit each message received to a UART at 1200 baud so it can be seen on a terminal.

When a message is received a bit is set on the channel status register CANSTCH (see Figure 13). We can poll this bit to determine when a message has been received. After processing the message we can then set this bit to zero to be ready for the next message.

**CAN Message Object Status Register**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLCW</strong></td>
<td><strong>TXOK</strong></td>
<td><strong>RXOK</strong></td>
<td><strong>BERR</strong></td>
<td><strong>SERR</strong></td>
<td><strong>CERR</strong></td>
<td><strong>FERR</strong></td>
<td><strong>AERR</strong></td>
</tr>
</tbody>
</table>

**Figure 13**
The channel status register.
A main program which looks at RXOK in the CANSTCH register and resends the received CAN message over the UART line looks like this:

```c
EX0 = 1;      //Turn on external interrupt 0
IT0 = 1;      //Set interrupt 0 for edge trigger
EA = 1;       //Global interrupt enable.
CANPAGE = (0 << 4);                    // CHNB=0x00; select channel 0
while(1)
{if((CANSTCH & MSK_CANSTCH_RxOk) == 0) //MAK_CANSTCH_RxOK = 0x20
{for (i=0; i<8; i++)                 //Put message in can_data[i]
    can_data[i] = CANMSG;             // Note autoindexing on CANMSG
    SendUARTMessage(can_data);
}
}
```

The SendUARTMessage function must set up the UART for the proper baud rate and transmit the 8 bytes of data in can_data. If the message is not already in ASCII, this function should convert the message to ASCII hex characters and send that.

To transmit data, we need an interrupt function for external interrupt 0. Each time the user pushes the interrupt 0 button this program will send on 8-byte message on the CAN bus. The interrupt 0 function will look like this:

```c
// This function is triggered by the push button on P3.2. It counts
// up on port 2 and sends an ASCII message on the CAN bus.
void SendData() interrupt 0 using 1
{char save_canpage;
char mData[8] = "HelloMom"; // 8 character message to be sent
save_canpage = CANPAGE;     //Save the old CAN page number
SendCANMessage(mData);      //Send the message on the CAN bus
P2++;                       //Count up on P2 to indicate we got here
CANPAGE = save_canpage;     //Restore the old CAN page
}
```

```c
// This function is triggered by the push button on P3.2. It counts
// up on port 2 and sends an ASCII message on the CAN bus.
void SendData() interrupt 0 using 1
{char save_canpage;
char mData[8] = "HelloMom"; // 8 character message to be sent
save_canpage = CANPAGE;     //Save the old CAN page number
SendCANMessage(mData);      //Send the message on the CAN bus
P2++;                       //Count up on P2 to indicate we got here
CANPAGE = save_canpage;     //Restore the old CAN page
}
```

```c
void SendCANMessage(char mData[])
{char i;                   // can_data index
CANPAGE = (1 << 4);       // select channel 1
CANCONCH = CH_DISABLE;    // reset channel 1 configuration
for (i=0; i<8; i++)       //Load the message from can_data[i]
    CANMSG = mData[i];
CANCONCH |= DLC_MAX;      //transmit 8 bytes (DLC_MAX = 8)
CANCONCH |= CH_TxENA;     //Enable transmission CH_TxENA = 0x40
CANEN2 |= (1 << 1);       //Enable channel 1
CANSTCH=0x00;             //reset channel 1 status
CANGIT = 0;               //Reset CAN interrupts
CANCONCH &= ~CH_TxENA;    //Disable transmission
}
```

Note that the CAN controller will continue to send whatever message is in the buffer as long as it is enabled (CANCONCH |= CH_TxENA). We therefore disable transmission at the end of the function so that only one message goes out. (The CAN controller does not abort a message if it is disabled in mid-message.)

The complete program is given on the following pages.
/**
* CANSftWrPrj.c
* This program demonstrates the use of the CAN Controller
* for the AT89C51CC03 processor running with a 12MHz crystal
* is the standard mode (not x2). The CAN bus runs at
* 500 KBaud.
* A pushbutton is assumed on P3.2 for use with external int0
* Each time the user pushes this button a CAN message is
* sent on the CAN bus on Channel 1 with ID 0x001.
* The software also polls the receiver CAN line for incoming
* CAN messages on Channel 0 with id 0x123. Each time a
* message is received it is retransmitted at 19,200 baud on
* the serial port as an 8 byte ASCII message. This message
* can be viewed on a serial port terminal with 8-bits of data,
* no parity, and no flow control at 19,200 baud.
*/
#include "t89c51cc01.h"
// Baud rate and bit timing parameter constants
#define BT1 0x00
#define BT2 0x02
#define BT3 0x2B

void SendCANMessage(char mData[]);
void SendUARTMessage(char mData[]);
void main(void)
{
  unsigned char num_channel, num_data, i;
  char can_data[8];
  //CANGCON = CAN General Control Register
  CANGCON |= MSK_CANGCON_GRES;   //Sets bit 0 in CANGON to reset CAN
  // Reset all mailboxes */
  for (num_channel = 0; num_channel < 15; num_channel++)
  {
    CANPAGE = num_channel << 4;  // CANGCON = CAN General Control Register
    CANGCON |= MSK_CANGCON_GRES;  //Sets bit 0 in CANGON to reset CAN
    // Reset all mailboxes */
    CANPAGE = num_channel << 4;
    //CANCONCH = CAN Message Object Control and DLC Register
    CANCONCH = CH_DISABLE;  //Disables transmission and reception messages
    CANSTCH = 0;  //CAN Message Object Status Register
    CANIDT1 = 0;  //CAN Identifier Tag Register 1
    CANIDT2 = 0;  //Set all of the identifier tags to 0
    CANIDT3 = 0;
    CANIDT4 = 0;
    CANIDM1 = 0;  //CAN Identifier Mask Registers 1
    CANIDM2 = 0;  //0 masks out ID bits so ID disabled
    CANIDM3 = 0;
    CANIDM4 = 0;
    for (num_data = 0; num_data < 8; num_data++)
    {
      CANMSG = 0;  //CAN Message Data Register. MSG register address is
      // auto incremented so this makes all 8 messages = 0
    }
    // bit timing
    CANBT1 = BT1;  //BT1 is the baud rate generator. BRP=0x00;
    CANBT2 = BT2;  //BT2 has tSJW and tPRS
    CANBT3 = BT3;  //BT3 has tPHS1 and tPHS2
    CANGCON |= MSK_CANGCON_ENA; //Start the CAN Controller state machine.
  }
  // Channel 0 initialization
  CANPAGE = (0 << 4);  // CHNB=0x00; select channel 0. Bits 4 and 5 = 00
  CANSTCH = 0x00;  // reset channel status
  CANCONCH = CH_DISABLE; // reset control and dlc register
  //Channel 0: identifier = 11bits. CANIDT=0x123 = 001 0010 0011
  CANIDT1 = 0x24;  //CANIDT1 is bits 3 to 10 of the ID = 001 0010 0011
  CANIDT2 &== 0x80;  //CANIDT2 has bits 0 - 2 in positions 5, 6, and 7 = 011
  CANIDT2 |= 0x60;
  // Channel 0: mask = 11bits. 0x7F0 = 111 1111 0000
}
CANIDM1 = 0xFE;     //CANIDM1 has bits 3 to 10 = 1111 1110
CANIDM2 &= ~0xE0;   //CANIDM2 has bits 0-2 in positions 5, 6, and 7 = 000
CANIDM4 = 0;       //Disables remote transmission & makes 0 higher priority than 1
CANIDT4 &=~0x04;    //Set Remote Transmission Request to 0 in IDT4
//Disables remote transmission and makes 0 higher priority than 1
//Channel 1 init
CANPAGE = (1 << 4);    //Bits 4-5 in CANPAGE = 1 for Channel 1
CANSTCH = 0x00;        //Reset the channel status
CANCONCH = CH_DISABLE; // reset control and dlc register
/* Channel 1: identifier = 11bits. CANIDT=0x401 = 100 0000 0001*/
CANIDT1 = 0x00;        //CANIDT1 is bits 3 to 10 of the ID = 0000 0000
CANIDT2 &= ~0xC0;       //CANIDT2 has bits 0-2 in positions 5, 6, and 7 = 001
EX0 = 1;      //Turn on external interrupt 0
IT0 = 1;      //Set interrupt 0 for edge trigger
EA = 1;       //Global interrupt enable.
CANPAGE = (0 << 4);                    // CHNB=0x00; select channel 0
while(1)
{
    if((CANSTCH & MSK_CANSTCH_RxOk) == 0) //MAK_CANSTCH_RxOK = 0x20
    {
        for (i=0; i<8; i++)                 //Put message in can_data[i]
            can_data[i] = CANMSG;            // Note autoindexing on CANMSG
            SendUARTMessage(can_data);
    }
}
} // This function is triggered by the push button on P3.2. It counts
// up on port 2 and sends an ASCII message on the CAN bus.
void SendData() interrupt 0 using 1
{
    char save_canpage;
    char mData[8] = "HelloMom";   //8 character message to be sent
    save_canpage = CANPAGE;      //Save the old CAN page number
    SendCANMessage(mData);      //Send the message on the CAN bus
    P2++;                       //Count up on P2 to indicate we got here
    CANPAGE = save_canpage;     //Restore the old CAN page
} //This function sends the message in mData out over the CAN
// network.
void SendCANMessage(char mData[])
{
    char i;                   // can_data index
    CANPAGE = (1 << 4);       // select channel 1
    CANCONCH = CH_DISABLE;    // reset channel 1 configuration
    for (i=0; i<8; i++)       //Load the message from can_data[i]
        CANMSG = mData[i];
    CANCONCH |= DLC_MAX;      //transmit 8 bytes (DLC_MAX = 8)
    CANCONCH |= CH_TxENA;     //Enable transmission CH_TxENA = 0x40
    CANEN2 |= (1 << 1);       //Enable channel 1
    CANSTCH=0x00;             //reset channel 1 status
    CANGIT = 0;               //Reset CAN interrupts
    CANCONCH &= ~CH_TxENA;    //Disable transmission
} //This function sends the message in mData over the UART
// at 1200 baud
void SendUARTMessage(char mData[])
{
    unsigned char i;
    SCON = 0x40;       // Mode 2, 8 bit uart transmit only, uses T2
    TMOD = 0x20;       // Turn off receive clock in T2CON -> use T1
    RCLK=0;            // Turn off transmit clock in T2CON -> use T1
    TH1=230;           //Baud rate = (fCrystal/12)/(32*(256 - (T1H))
    // If double clocked, multiply fCrystal by 2.
    TR1=1;             // 1200 baud @ 12 Mhz no double clock
    REN=0;             // TCON bit to start Timer 2
    // Transmit only

RI = 0;            // Clear the receive interrupt flag
for(i=0;i<8;i++)
    {TI = 0;
     SBUF = mData[i];
     i++;
     while (TI == 0);   // Wait for write to be done
    }
TI = 0;
SBUF = 0x0D;       //CR
while (TI == 0);   // Wait for write to be done
TI = 0;
SBUF = 0x0A;       //LF
while (TI == 0);   // Wait for write to be done
}

**CAN Bus Hardware**
The CAN bus requires an interface chip to match voltage levels and other electrical specifications to the processor. We will use the Atmel ATA6660 chip for this purpose. A typical circuit diagram is given in Figure 14. CN-1 is the power connector. CN-2 accepts the CAN signals from the processor (on P4.0 and P4.1). CN-3 is the CAN bus connector. Pins 1 and 3 typically go to a twisted pair. The jumper is an optional way of adding a 120 ohm terminating impedance to the bus.

![Figure 14](image-url)  
CAN bus hardware interface.
Grading:
This project will be done individually and a single grade will be given for each project. A total of 100 points is available for the project and will be awarded on the following basis:

<table>
<thead>
<tr>
<th>Points</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 points</td>
<td>Does your project transmit on the CAN bus.</td>
</tr>
<tr>
<td>20 points</td>
<td>Does your project receive on the CAN bus.</td>
</tr>
<tr>
<td>15 points</td>
<td>Finished product quality</td>
</tr>
<tr>
<td>20 points</td>
<td>Documentation of software</td>
</tr>
<tr>
<td>20 points</td>
<td>Documentation of hardware</td>
</tr>
</tbody>
</table>

At a minimum your hardware documentation must consist of a system diagram, a complete circuit diagram (with pin numbers), and a mechanical sketch of your project done to a level of detail such that another person in the class could build your project from your diagram. At a minimum your software documentation should consist of fully commented source code for all of the modules in your program and a pseudocode design with enough detail that another person in the class could duplicate the function of your software.