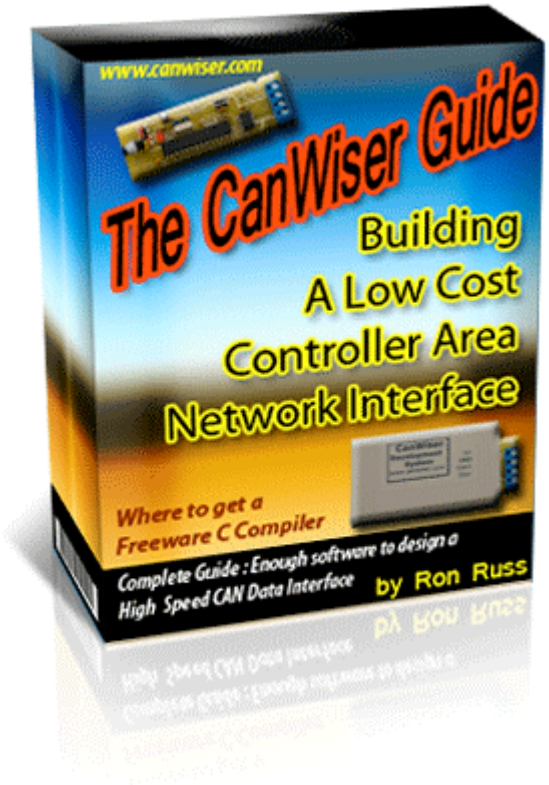


The CanWiser Guide
Building A Low Cost Controller Area Network Interface

Table of Contents

Chapter 1 - Introduction

The CanWiser Guide *Building A Low Cost Controller Area Network Interface* is a hands-on guide designed to quickly and cost effectively get you programming your own Controller Area Network (CAN) project in C. The PIC18F258 Flash Microcontroller from MicroChip is used along with the *High-Tech C Pro for the PIC18 MCU Family (Lite mode)* Freeware C Compiler for learning to program embedded microcontrollers with CAN. Since this is a hands on guide, all the necessary tools and hardware are shown to get you up and running in a short period of time.

The projects start with the basics and then advances to projects showing how to build your own custom CAN tool.

There are 3 ways to use this guide. First off you could just read through it to learn about the topics which will give you a basic understanding of using a CAN micro. The second way, which will require some extra time, is to build the basic circuitry (The CanWiser), get the USB interface, and then follow along. The recommended way is to purchase the CanWiser module and you'll be off and running in as little time as possible. The CanWiser is available either through the Emicros Ebay store at http://stores.ebay.com/emicros or at www.emicros.com.

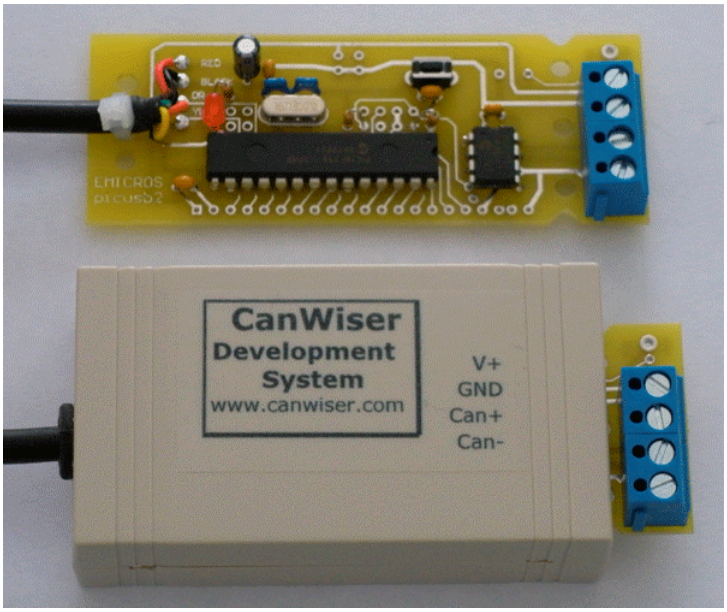
Chapter 2 – Getting Started With The Tools

This book is a hands-on guide to microcontrollers. Reading about microcontrollers is fine for reference or as an introduction to microcontrollers. But for the motivated reader that really wants to learn about microcontrollers, you have to have both hardware and software. Emulators usually cost too much for the non-professional so the challenge is to provide a cost effective system that allows software to be written quickly and tested on hardware

Now we need to get our tools ready to go.

Getting Started with the CanWiser

The CanWiser (shown below with optional plastic enclosure) used in this book is a complete, low-cost development system that contains a PIC 18F258 Flash Microcontroller from Microchip. The PIC is programmed with a bootloader that allows you to download and test your programs in a matter of seconds.



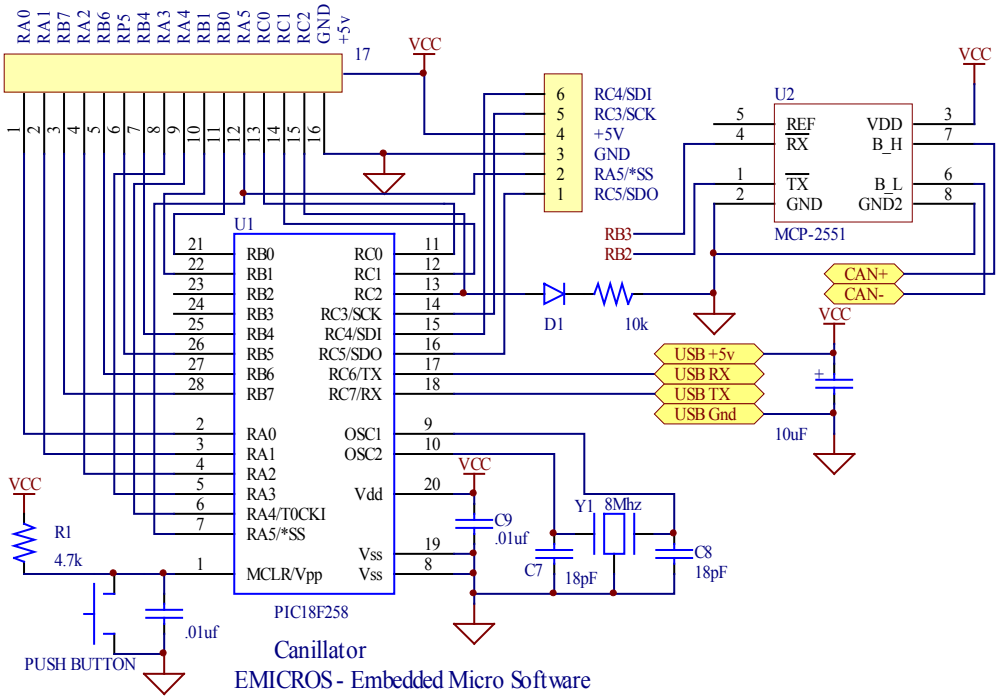
CanWiser Schematic

The following schematic diagram shows the CanWiser circuitry. The PIC18F258 contains a CAN peripheral controlling pins RB2(CANTX) and RB3(CANRX) which are connected to the MCP-2551 CAN transceiver IC. The output of the MCP-2551 is connected to the CAN bus through the external 'blue' connector on CanWiser.

The 8Mhz (megahertz) crystal clocks the micro but the High Speed PLL is enabled boosting the interal clock speed to 32Mhz.

Pin RC2 has an indicator LED connected to it which comes in handy for debugging or status information. The default program uses the LED to indicate if it is offline (solid on) or if online it will toggle indicating a new CAN message has been received.

The Reset PushButton is needed to but the device into download mode for programming from the PC. Once the button is push there is a 1 second window for starting the programming.



Connecting CanWiser to the PC

Before we get started we need to hook up the CanWiser to the PC. A USB interface is used for a high-speed serial communication port and also provides +5 volts for power. The serial communications is important for downloading programs from the PC using the interface program detailed in the next section. Once the programs are downloaded, the serial channel can be used for various functions such as reading data from the CanWiser or sending commands to it.

The first time you connect the CanWiser to your PC with the USB cable, Windows will detect it and need to install the driver for it. The Virtual Com Port (VCP) drivers cause the USB device on the CanWiser to appear as a COM port on the PC. The most current driver is available at www.ftdichip.com/vcp.htm. Click on the link and save the zipped file to a directory like c:\ftdi. Unzip the files to the same directory and when you plug in the CanWiser and Windows ask you where the driver is you can point it to this directory.

Note that when you connect the CanWiser to the PC using the USB cable, the onboard LED will turn on if you haven't overwritten the default program.

Pressing the reset switch will cause the micro to enter the reset state and the LED will remain off while the switch is pressed. The reset switch is important when downloading programs to the CanWiser.

Downloading Programs to CanWiser

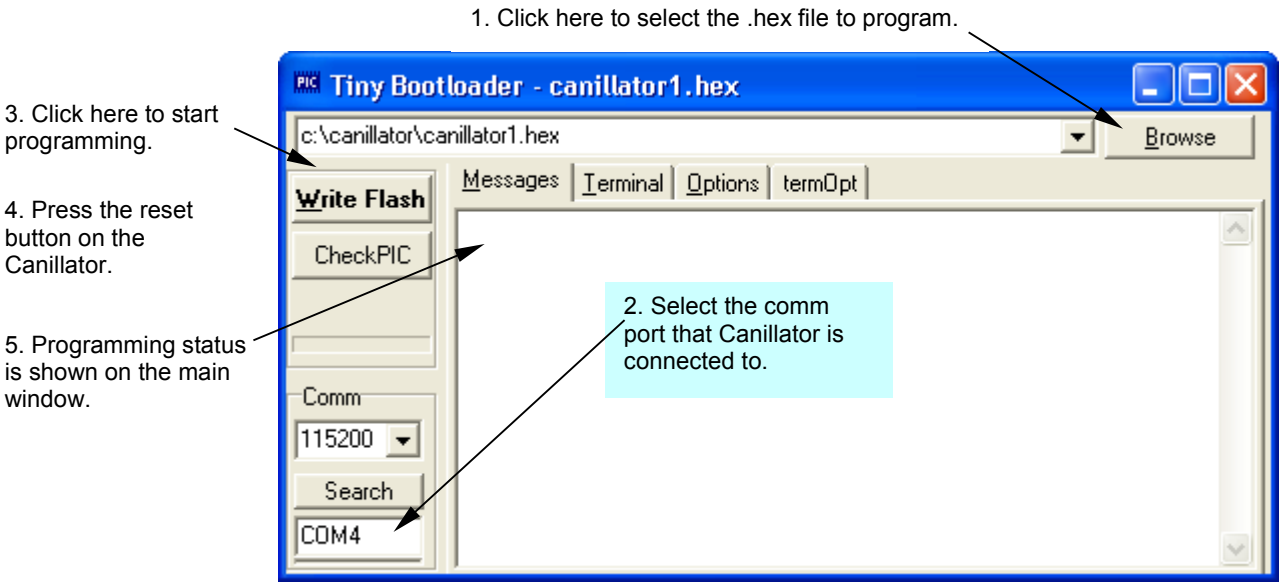
The CanWiser contains a bootloader that allows it to be programmed, or reflashed, from the PC. This small-embedded program executes only after a reset (caused by either a power-up or pressing the reset button) and looks for a command to be sent to it over the USB connection from the PC. If there are no commands after a short period of time, execution is transferred to your application. The CanWiser uses the "Tiny Boot Loader" which saves space and works rather well.

At reset, the micro loads the program counter with the address contained in the reset vector. This address usually contains the address of the first location to execute in your application. Instead, the address of the bootloader is placed in the reset vector (this is already provided) causing the bootloader to be executed at reset. The bootloader is a very small program that is placed in the upper portion of memory so that your application can use the remaining bulk of the program memory.

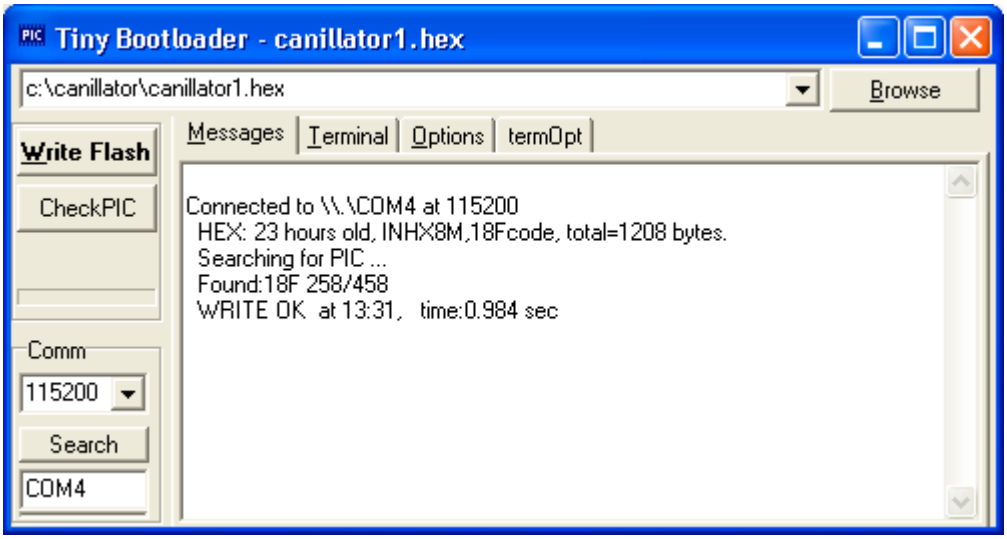
The bootloader executes for a very short period of time watching the serial channel for commands and if it does see any then it will jump to the real start of the application program.

In order for you to quickly develop programs and download them to the board, a PC downloader program is needed. This program reads the file containing your program and sends it over the COM port to CanWiser and it knows how to program itself. When the programming is complete, CanWiser immediately executes your program.

The CanWiser downloader program for the PC is available in the zipped file at www.emicros.com/tinybootloader.zip. Unzip the files to a directory such as c:\CanWiser and execute the tinybldWin.exe file.



The following shows the completed programming sequence.

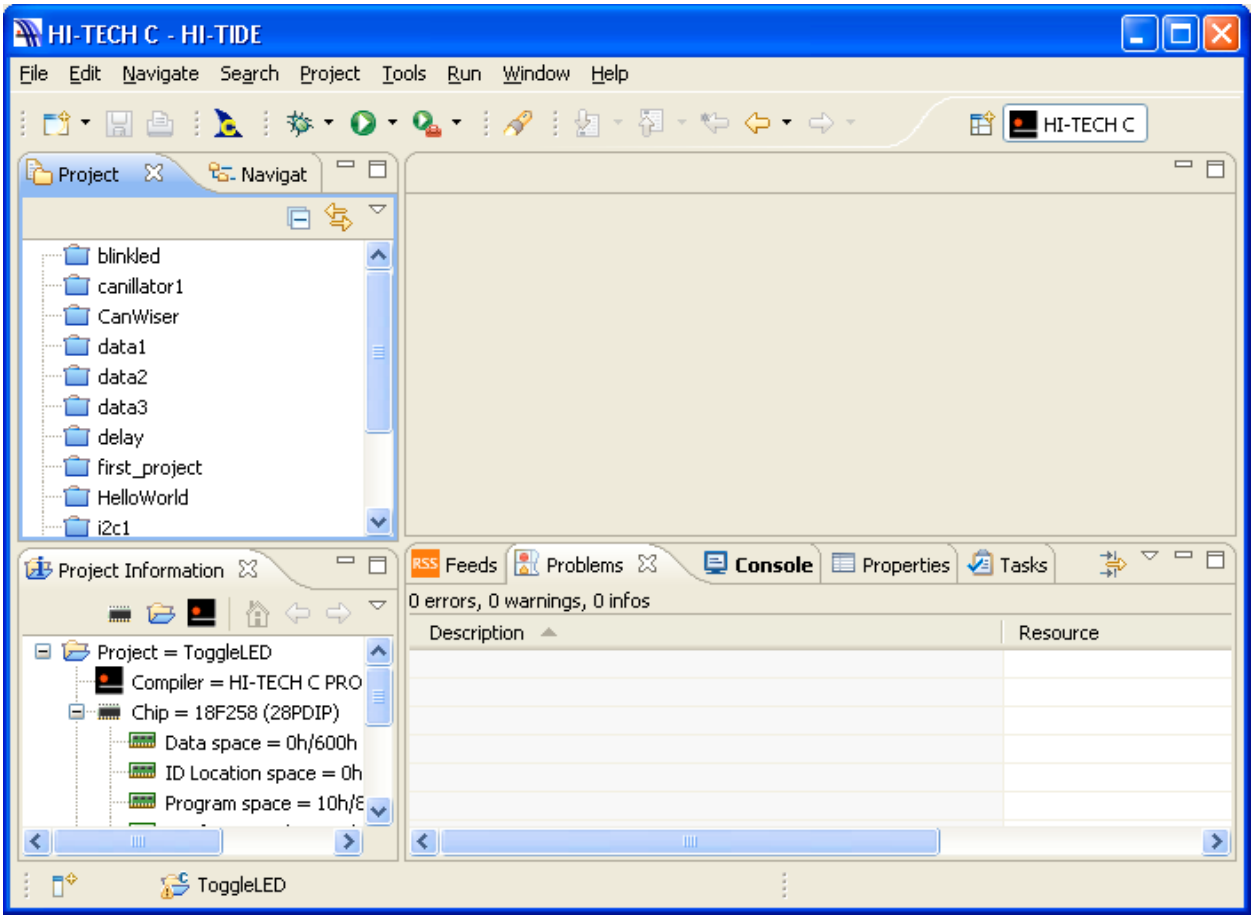


Getting the C Compiler

HI-Tech Software offers a free evaluation version of their PIC C compiler at microchip.hi-tech.com. Go to the HI-Tech web site at microchip.hi-tech.com to download the Lite (Freeware) version of the *HI-TECH C Pro for the PIC18 MCU Family*. During the installation process an option to download the Integrated Development Environment HI-TIDE will be given. Install this also.

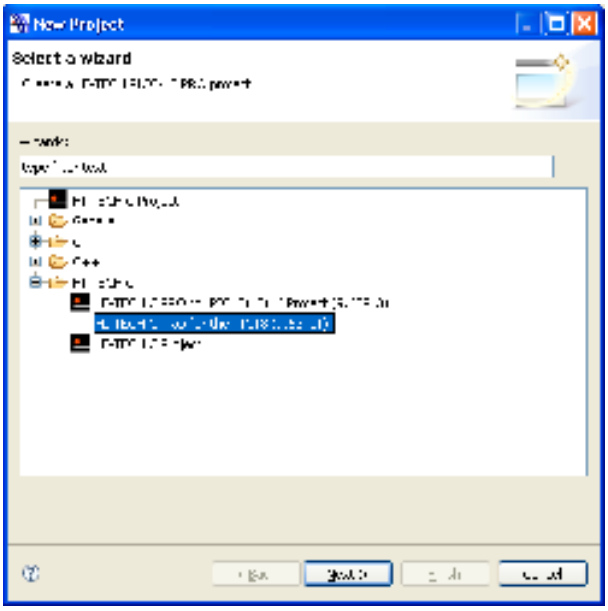
Note that as of this writing the compiler file is HCPIC18P-PRO-9.63PL1.exe and the HI-TIDE version is 3.15. Yours might be different.

After installation, launch HI-TIDE either automatically when the installation procedure exits or clicking on the icon on your desktop. The following is an example of the integrated development environment (IDE) and yours will be different since you don't have any projects yet.

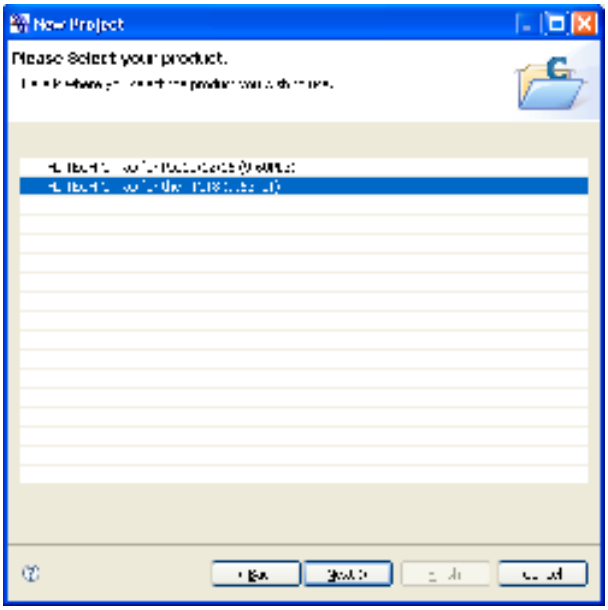


Starting A New Project

The first thing we want to do is create a new project. Click on File\New\Project, which opens the window shown here, followed by a click on HI-TECH C PRO for the PIC18 (9.63PL1) and finally click on the Next > button.

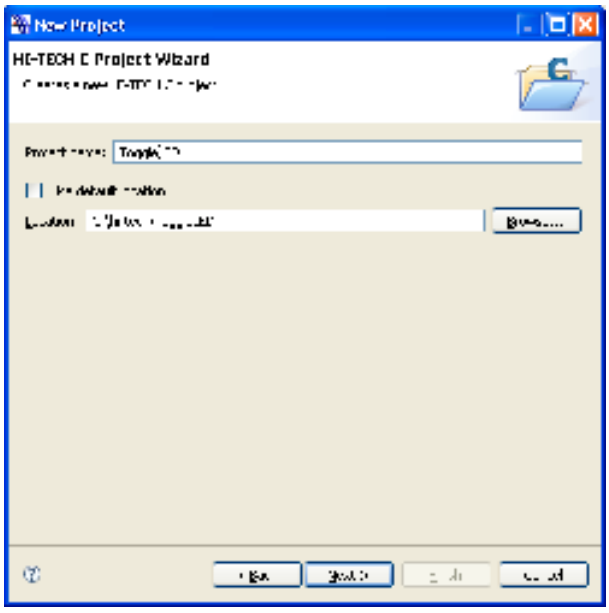


The next window needs you to click on the HI-TECH C PRO for the PIC18 (9.63PL1) product.

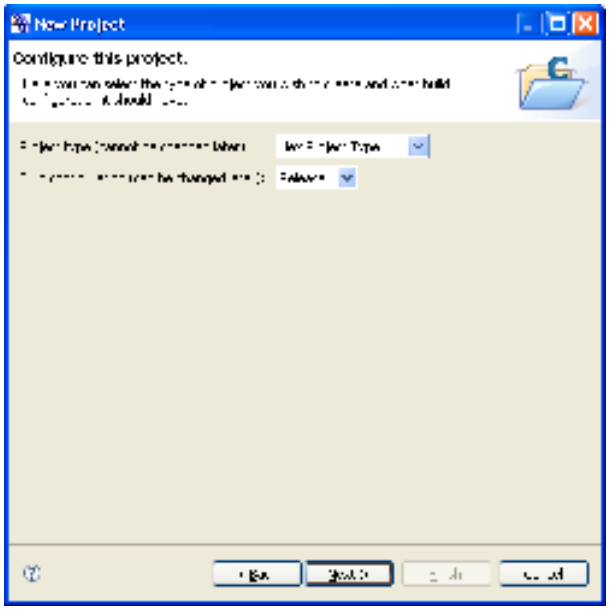


In the **Project name:** field enter a descriptive project name, uncheck the **Use default location** checkbox, and enter a directory location where you want to store this project. Creating a directory such as c:\hi-tech to store your projects under is an easy way to keep them easy to find.

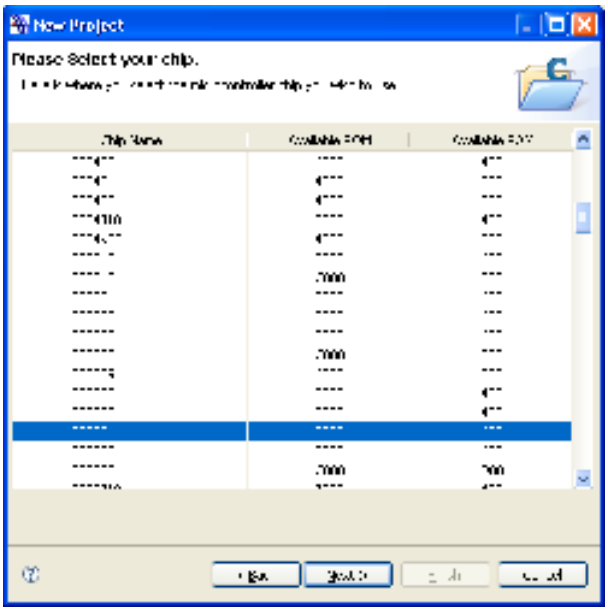
Hit the **Next >** button to go to the next selection.



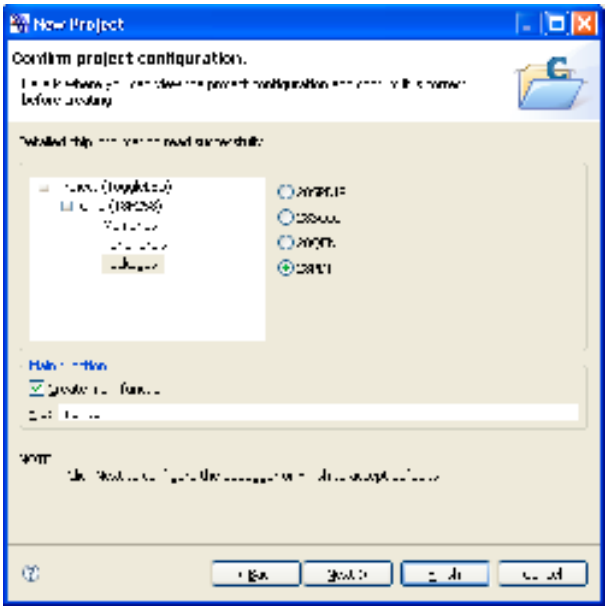
Accept the default items on this screen and go on the next one by hitting the **Next >** button.



Scroll down to the 18F258 chip and click on it to select it. Hit the **Next >** button.



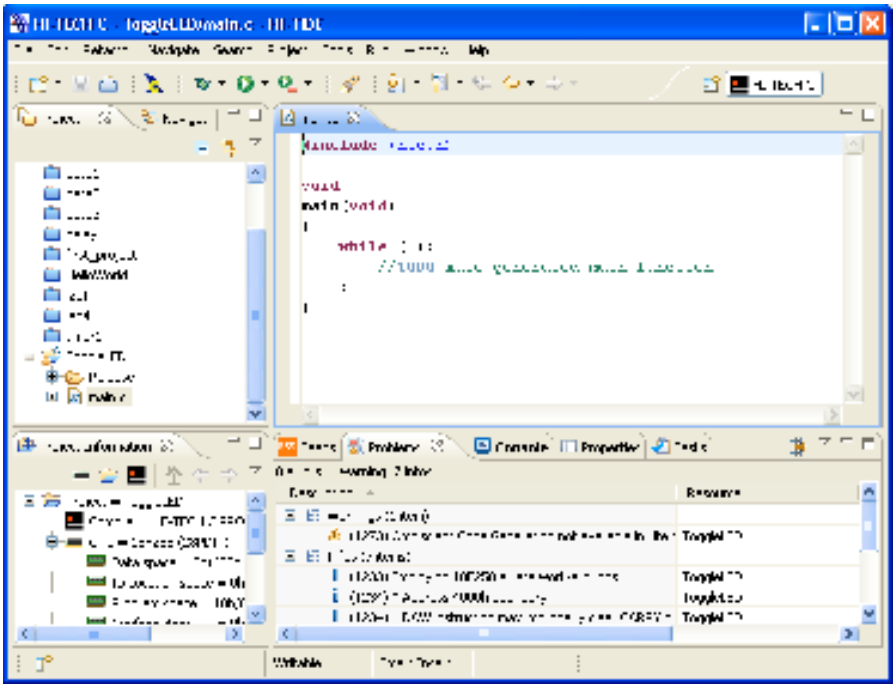
Select the 28PDIP package and hit the **Finish** button.



The project wizard will create the ToggleLED project as shown in the Project tab.

Double click on the main.c file under the ToggleLED project to open it.

In order to



Chapter 3 - CAN Basics

In this chapter the basics for dealing with the Controller Area Network are explored.

CAN Initialization

In this section we initialize the CAN module.

Function : ecan_set_mode()

The **ecan_set_mode()** function is used to request a change to the operating mode of the CAN module. A request method is needed in case the module is in the process of transmitting or receiving any messages. The following modes can be requested;

- Configuration Mode
- Listen Only Mode
- Loopback Mode
- Disable Mode
- Normal Mode

The requested mode is passed to the function and should be one of the following values that are defined in the ecan.h header file.

```
#define CONFIG_MODE       (0x80)
#define LISTEN_MODE       (0x60)
#define LOOPBACK_MODE (0x40)
#define DISABLE_MODE    (0x20)
#define NORMAL_MODE    (0x00)
```

The upper 3 bits of the CANCON register requests the mode and the value in mode is or'd into the current CANCON register. Once the CANCON is written to, the program what for the CANSTAT register to show that the mode has changed.

The x variable is used to prevent the while loop from locking up. Just in case it does the function will notify the calling routine by returning the ERROR value. The _delay() function is used to give the module some time to change.

The function returns OK when the mode changes.

```
UINT8 ecan_set_mode( UINT8 mode )
{
   UINT8 x;
   CANCON = mode | (CANCON & 0x1f);
   x = 0;
   while( (CANSTAT & 0xe0) != mode )
   {
      x++;
      if( x > 10 )
      {
         return( ERROR );
      }
      _delay( 1000 );
   }
   return( OK );
}
File : ecan.c     Prototype : ecan.h
```

Function : ecan_set_baud()

The **ecan_set_baud()** function is used to set the 3 baud rate registers BRGCON1, BRGCON2, and BRGCON3 to values achieving the desired CAN baud rate. The function is passed a value selecting the baud rate as shown below. Notice that there is a #define that reduces the amount of selections to save memory space since in most cases just 1 baud rate is needed.

```
void ecan_set_baud( UINT8 baudrate )
{
  #ifdef USE_FULL_BAUDRATE_TABLE
   if( baudrate == 0 ) { BRGCON1=0x00; BRGCON2=0xb8; BRGCON3=0x05; return; }// 1 mbit
   if( baudrate == 1 ) { BRGCON1=0x00; BRGCON2=0xb8; BRGCON3=0x07; return; }// 889
   if( baudrate == 2 ) { BRGCON1=0x01; BRGCON2=0xb0; BRGCON3=0x02; return; }// 800
   if( baudrate == 3 )  { BRGCON1=0x01; BRGCON2=0xb8; BRGCON3=0x02; return; }// 727
   if( baudrate == 4 )  { BRGCON1=0x01; BRGCON2=0xb8; BRGCON3=0x03; return; }// 667
   if( baudrate == 5 )  { BRGCON1=0x01; BRGCON2=0xb8; BRGCON3=0x05; return; }// 500
   if( baudrate == 6 )  { BRGCON1=0x03; BRGCON2=0xb8; BRGCON3=0x05; return; }// 250
   if( baudrate == 7 )  { BRGCON1=0x05; BRGCON2=0xb8; BRGCON3=0x05; return; }// 167
   if( baudrate == 8 )  { BRGCON1=0x07; BRGCON2=0xb8; BRGCON3=0x05; return; }// 125
   if( baudrate == 9 )  { BRGCON1=0x09; BRGCON2=0xb8; BRGCON3=0x05; return; }// 100
   if( baudrate == 10 ) { BRGCON1=0x0b; BRGCON2=0xb8; BRGCON3=0x05; return; }// 83
   if( baudrate == 11 ) { BRGCON1=0x0f; BRGCON2=0xb8; BRGCON3=0x05; return; }// 62.5
   if( baudrate == 12 ) { BRGCON1=0x13; BRGCON2=0xb8; BRGCON3=0x05; return; }// 50
   if( baudrate == 13 ) { BRGCON1=0x17; BRGCON2=0xb8; BRGCON3=0x05; return; }// 41.67
   if( baudrate == 14 ) { BRGCON1=0x1d; BRGCON2=0xb8; BRGCON3=0x05; return; }// 33
   if( baudrate == 15 ) { BRGCON1=0x1f; BRGCON2=0xb8; BRGCON3=0x05; return; }// 31.25
   if( baudrate == 16 ) { BRGCON1=0x27; BRGCON2=0xb8; BRGCON3=0x05; return; }// 25
   if( baudrate == 17 ) { BRGCON1=0x31; BRGCON2=0xb8; BRGCON3=0x05; return; }// 20
   if( baudrate == 18 ) { BRGCON1=0x3f; BRGCON2=0xbf; BRGCON3=0x07; return; }// 10
   if( baudrate == 19 ) { BRGCON1=custom_brgcon[0];
                          BRGCON2=custom brgcon[1];
                          BRGCON3=custom_brgcon[2];  return; } // Custom
  #else
   if( baudrate == 5 )  { BRGCON1=0x01; BRGCON2=0xb8; BRGCON3=0x05; return; }// 500
  #endif
}
```

Function : ecan.c Prototype : ecan.h

Function : ecan_init()

The **ecan_init()** function initializes the CAN module by

1. Configuring the port pins
2. Changing the mode to CONFIG MODE
3. Setting the baud rate
4. Setting any other registers
5. Changing the mode to NORMAL so the module is ready to transmit and receive messages.

```
void ecan_init( void )
{
   /* ---------------------------------------
   // Set the data direction for the CAN pins
   // ---------------------------------------*/
   TRISB3 = 1;  /* CAN rx on rb3 */
   TRISB2 = 0;  /* CAN tx on rb2 */

   ecan_set_mode( CONFIG_MODE );
   ecan_set_baud( DEFBAUD );
   ecan_set_mode( NORMAL_MODE );
}
```

File : ecan.c Prototype : ecan.h

CAN Transmit Messages

The PIC18F258 has 3 transmit message buffers for sending messages. Messages can range in length from 0 to 8 bytes and support both 11 bit Standard and 29 bit Extended identifiers.

Data Structure : CAN_MESSAGE

Before we can transmit a CAN message, we need a way to deal with setting the id and data in the message. Instead of having the application deal directly with writing to the message registers, we will use a structure to hold the values.

The CAN_MESSAGE structure shown contains the following;

id_type This is set to STANDARD or EXTENDED
len Message length 0 to 8 bytes
dta[8] Array of 8 values
union{}id Standard or Extended ID values

```
typedef struct {
   UINT8 id_type;
   UINT8 len;
   UINT8 dta[8];
   union {
   UINT16 sid;
   UINT32 eid;
   } id;
} CAN_MESSAGE;
```

File : ecan.h

In the file ecan.c the transmit buffers are allocated. Notice that in order to save space there is a #define around the transmit 1 and 2 buffers.

```
CAN_MESSAGE TxMessage0;
#ifdef USE TX MESSAGE1
  CAN_MESSAGE TxMessage1;
#endif
#ifdef USE TX MESSAGE2
  CAN MESSAGE TxMessage2;
#endif
```

File : ecan.c

In order for the application to get access to the transmit buffers, the buffers are externed.

If the 2 other transmit buffers are needed, the comments on line 1 and/or 2 can be removed.

```
//#define USE_TX_MESSAGE1
//#define USE_TX_MESSAGE2

extern CAN_MESSAGE TxMessage0;
#ifdef USE TX MESSAGE1
 extern CAN_MESSAGE TxMessage1;
#endif
#ifdef USE TX MESSAGE2
 extern CAN_MESSAGE TxMessage2;
#endif
```

File : ecan.h

Functions : ecan_send_Tx()

For each of the 3 transmit message buffers we have a send message function as shown in the following 3 figures.

The TXBnDLC contains the length of the message to send and can be from 0 to 8 bytes. Notice that the length is not boundary check (i.e. less than or equal to 8). This could be added for robustness but to save space it has not been added.

TXBnB0 through TXBnB7 contains the data bytes to send.

The message type is checked and the associated id registers are set for 11 or 29 bit id's.

Setting bit 3 in TXBnCON tells the module to send it.

```
void ecan_send_Tx0( void )
{
   TXB0DLC = TxMessage0.len;
   TXB0D0  = TxMessage0.dta[0];
   TXB0D1  = TxMessage0.dta[1];
   TXB0D2  = TxMessage0.dta[2];
   TXB0D3  = TxMessage0.dta[3];
   TXB0D4  = TxMessage0.dta[4];
   TXB0D5  = TxMessage0.dta[5];
   TXB0D6  = TxMessage0.dta[6];
   TXB0D7  = TxMessage0.dta[7];

   if( TxMessage0.id_type == STANDARD )
   {
      TXB0SIDH = (UINT8)(TxMessage0.id.sid >> 3);
      TXB0SIDL = (UINT8)(TxMessage0.id.sid << 5);
   }
   else if( TxMessage0.id_type == EXTENDED )
   {
      /* need to add code here */
   }

   TXB0CON |= 0x08;   /* send the message */
}
File : ecan.c     Prototype : ecan.h
```

```
#ifdef USE TX MESSAGE1
void ecan_send_Tx1( void )
{
   TXB1DLC = TxMessage1.len;
   TXB1D0  = TxMessage1.dta[0];
   TXB1D1  = TxMessage1.dta[1];
   TXB1D2  = TxMessage1.dta[2];
   TXB1D3  = TxMessage1.dta[3];
   TXB1D4  = TxMessage1.dta[4];
   TXB1D5  = TxMessage1.dta[5];
   TXB1D6  = TxMessage1.dta[6];
   TXB1D7  = TxMessage1.dta[7];
   if( TxMessage1.id_type == STANDARD )
   {
      TXB1SIDH = (UINT8)(TxMessage1.id.sid >> 3);
      TXB1SIDL = (UINT8)(TxMessage1.id.sid << 5);
   }
   else if( TxMessage1.id_type == EXTENDED )
   {
      /* need to add code here */
   }
   TXB1CON |= 0x08;      /* send the message */
}
#endif
File : ecan.c     Prototype : ecan.h
```

```
#ifdef USE_TX_MESSAGE2
void ecan_send_Tx2( void )
{
   TXB2DLC = TxMessage2.len;
   TXB2D0  = TxMessage2.dta[0];
   TXB2D1  = TxMessage2.dta[1];
   TXB2D2  = TxMessage2.dta[2];
   TXB2D3  = TxMessage2.dta[3];
   TXB2D4  = TxMessage2.dta[4];
   TXB2D5  = TxMessage2.dta[5];
   TXB2D6  = TxMessage2.dta[6];
   TXB2D7  = TxMessage2.dta[7];
   if( TxMessage2.id_type == STANDARD )
   {
      TXB2SIDH = (UINT8)(TxMessage2.id.sid >> 3);
      TXB2SIDL = (UINT8)(TxMessage2.id.sid << 5);
   }
   else if( TxMessage2.id_type == EXTENDED )
   {
      /* need to add code here */
   }
   TXB2CON |= 0x08;/* send the message */
}
#endif
File : ecan.c     Prototype : ecan.h
```

Program : Transmit CAN Message Program 1

This simple program shows the basics for sending a CAN message from a background loop. The _delay() function provides a rough timing delay.

```
#include <htc.h>
#include <stdio.h>
#include "myheader.h"
#include "ecan.h"
#include "serial.h"
//-----------------------------------------------------
// file:        CanWiser1.c
// description: This is the first CanWiser project
// author:         Emicros
//-----------------------------------------------------
char sstr[40];
/*-----------------------------------------------------
** function:    InterruptServiceRoutine
** description: This is just a place holder for now
**              and isn't used in the program.
**-----------------------------------------------------*/
void interrupt InterruptServiceRoutine( void )
{
        if( (TMR1IE) && (TMR1IF) )
        {
                TMR1IF = 0;
        }
}
/*-----------------------------------------------------
** function:    main()
** description: This is the main function.
**-----------------------------------------------------*/
void main( void )
{
   TRISC = 0b11111011;
   SetupCommPort( BAUD9600 );
   sprintf( sstr, "Initializing CAN #1 ... " );
   CommPortSendString( sstr );
   ecan_init();
   TxMessage0.len    = 8; /* set the message length */
   TxMessage0.dta[0] = 0x01;/* set the data         */
   TxMessage0.dta[1] = 0x02;
   TxMessage0.dta[2] = 0x03;
   TxMessage0.dta[3] = 0x04;
   TxMessage0.dta[4] = 0x05;
   TxMessage0.dta[5] = 0x06;
   TxMessage0.dta[6] = 0x07;
   TxMessage0.dta[7] = 0x08;
   TxMessage0.id_type = STANDARD;
   TxMessage0.id.sid = 0x100;
   while( 1 )
   {
      PORTC = 0b00000100;
      ecan_send_Tx0();
      _delay( 100000 );
      PORTC = 0b00000000;
      _delay( 100000 );
      TxMessage0.dta[7]++;
    }
}
```

File : main.c

CAN Receive Messages

The PIC18F258 has 2 receive message buffers. Messages can range in length from 0 to 8 bytes and support both 11 bit Standard and 29 bit Extended identifiers. As in the case with transmitting messages, the data associated with receiving messages uses the same data structure.

Functions : ecan_setup_Rx0() and ecan_setup_Rx1()

The ecan_setup_Rx0() and ecan_setup_Rx1() functions take a single 16 bit value used for the 11 bit id and sets the receive. This is really only useful if the acceptance mask and filters are configured. In our case we will receive every message and provide filtering on it if necessary.

Notice that the 2nd receive buffer isn't used unless USE_RX_MESSAGE1 is defined.

```
void ecan_setup_Rx0( UINT16 id )
{
   RXB0CON = 0b00000000;

   RXB0SIDH = (UINT8)(id >> 3);
   RXB0SIDL = (UINT8)(id << 5);
}
#ifdef USE_RX_MESSAGE1
void ecan_setup_Rx1( UINT16 id )
{
   RXB1CON = 0b00000000;

   RXB1SIDH = (UINT8)(id >> 3);
   RXB1SIDL = (UINT8)(id << 5);
}
#endif

File : ecan.c     Prototype : ecan.h
```

Functions : ecan_get_Rx0() and ecan_get_Rx1()

When a message is available in either receive buffer 0 or 1, the functions ecan_get_Rx0() and ecan_get_Rx1() are used to copy the contents to the RxMessage data structures.

```
void ecan_get_Rx0( void )
{
   UINT16 mid;

   RxMessage0.len    = RXB0DLC;
   RxMessage0.dta[0] = RXB0D0;
   RxMessage0.dta[1] = RXB0D1;
   RxMessage0.dta[2] = RXB0D2;
   RxMessage0.dta[3] = RXB0D3;
   RxMessage0.dta[4] = RXB0D4;
   RxMessage0.dta[5] = RXB0D5;
   RxMessage0.dta[6] = RXB0D6;
   RxMessage0.dta[7] = RXB0D7;

   if( (RXB0SIDL & 0x08) == 0x08 )
   {
   }
   else
   {
      mid = RXB0SIDH<<3;
      mid |= (RXB0SIDL>>5);
      RxMessage0.id.sid = mid;
   }
}

#ifdef USE_RX_MESSAGE1
void ecan_get_Rx1( void )
{
   UINT16 mid;

   RxMessage1.len    = RXB1DLC;
   RxMessage1.dta[0] = RXB1D0;
   RxMessage1.dta[1] = RXB1D1;
   RxMessage1.dta[2] = RXB1D2;
   RxMessage1.dta[3] = RXB1D3;
   RxMessage1.dta[4] = RXB1D4;
   RxMessage1.dta[5] = RXB1D5;
   RxMessage1.dta[6] = RXB1D6;
   RxMessage1.dta[7] = RXB1D7;

   if( (RXB1SIDL & 0x08) == 0x08 )
   {
   }
   else
   {
      mid = RXB1SIDH<<3;
      mid |= (RXB1SIDL>>5);
      RxMessage1.id.sid = mid;
   }
}
#endif
File : ecan.c     Prototype : ecan.h
```

Functions : ecan_setup_Rx0() and ecan_setup_Rx1()

The ecan_check_receive_flags() function is used to see if a new message has arrived. This function will return a value equal to NO_NEW_MESSAGE if both receive message buffers are empty. If either message is full, the associated bit is set so that in case the program is busy it will get an indication of both being full when it checks.

```
UINT8 ecan_check_receive_flags( void )
{
   UINT8 rval = NO_NEW_MESSAGE;

   if( RXB0IF == 1 )
   {
      ecan_get_Rx0();
      RXB0FUL = 0;
      RXB0IF = 0;
      rval |= NEW_MESSAGE_R0;
   }

   #ifdef USE_RX_MESSAGE1
    if( RXB1IF == 1 )
    {
      ecan_get_Rx1();
      RXB1FUL = 0;
      RXB1IF = 0;
      rval |= NEW_MESSAGE_R1;
    }
   #endif
   return( rval );
}
```

File : ecan.c Prototype : ecan.h

CAN Receive Interrupt

We will now use an interrupt in order to check for received messages instead of polling for them. This is shown in the following program.

In the interrupt service routine the Receive Buffer flags RXB0IF and RXB1IF are checked to see if a message has been received. If one has, then as an indicator the LED is toggled using the RC2 = !RC2 statement. The message is copied to the RAM buffer and the buffer full and interrupt flags are cleared.

The message id received is sent out the serial port indicating what id was received. In later programs the data will be sent also.

The CAN Receive Buffer 0 Interrupt Enable bit 0 must be set in the PIE3 register. This is set in the main function before interrupts are enabled as shown in the example code. Note that only the important code is shown.

```
void interrupt InterruptServiceRoutine( void )
{
   if( RXB0IF )
   {
      RC2 = !RC2;
      ecan_get_Rx0();
      RXB0FUL = 0;
      RXB0IF = 0;
      CommPortSendHex16( RxMessage0.id.sid );
      CommPortSendByte( 10 );
      CommPortSendByte( 13 );
   }
   #ifdef USE_RX_MESSAGE1
   if( RXB1IF )
   {
      RC2 = !RC2;
      ecan_get_Rx1();
      RXB1FUL = 0;
      RXB1IF = 0;
      CommPortSendHex16( RxMessage1.id.sid );
      CommPortSendByte( 10 );
      CommPortSendByte( 13 );
   }
   #endif
}

void main( void )
{
  /* enable receive buffer 0 interrupt  */
  PIE3 = 0xb00000001;

  /* enable global and peripheral interrupts */
  INTCON  = 0b11000000;

  while( 1 )
  {
  }
}
```

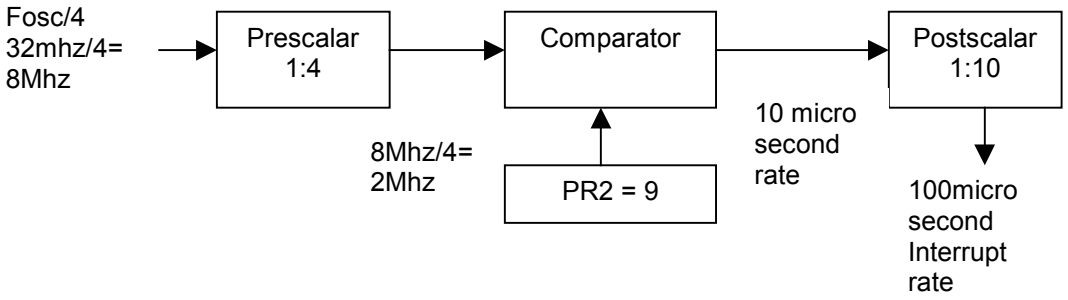
File : main.c

CAN Receive Message Time Stamping

The PIC18F258 has the ability to time stamp a received using one of the Capture/Compare Modules but I prefer to use a timer interrupt triggering every 100 microseconds instead. This provides an absolute time based system instead of a relative one and is easier to implement.

Timer2 : 100 Microsecond Interrupt

Timer2 is clocked by Fosc/4 which in the case of CanWiser is 32,000,000/4 or 8Mhz. This frequency is feed into a Prescaler that can divide it down by one of 3 value; 1:1, 1:4, or 1:16. Dividing by 4 results in a 2 Mhz clock that feeds the comparator whick is set to (10-1) or 9 resulting in a 10 microsecond rate. The final Postscalar is set to 10 resulting in a 100 microsecond interrupt rate.



The initialization code that is added to the main() function to setup the timer 2 is shown here.

The timer 2 enable bit (1) in PIE1 must be set also.

```
void main( void )
{
  /* ---------------------------------------------
  // Set the Timer 2 Control Register
  // Bit 7   unused=0    unused bit
  // Bit 6-3 0bx1001xxx  Postscale divide by (10-1)
  // Bit 2   TMR2ON = 0  Turn on Timer 2
  // Bit 1/0 T2CKPS1:T2CKPS0=01 Prescale (/4)
  // ---------------------------------------------*/
  T2CON   = 0b01001101;
  PR2 = 20-1;         // 100 microsecond int rate
  PIE1 = 0b00000010; // bit1 enables timer 2 int
```

File : main.c

The interrupt service routine needs software to check the timer 2 interrupt flag, reset the flag, and increment the time base as shown.

```
/* ---------------------------------------------
 * This timer is set to trigger every 100
 * microseconds to provide receive message
 * time stamping.
 * ---------------------------------------------*/
  if( TMR2IF )
  {
    IncrementTime();
    TMR2IF = 0;
  }
```

File : main.c

Function : IncrementTime()

The IncrementTime() function increments the 4 bytes rtime3 to rtime0 to keep track of time. These 4 bytes are allocated as UINT8 variables at the beginning of main.c and also initialized to 0 in the initialization portion of the main() function.

```
UINT8 rtime3, rtime2, rtime1, rtime0;

void IncrementTime( void )
{
   rtime3++;
   if( rtime3 == 0 )
   {
      rtime2++;
      if( rtime2 == 0)
      {
         rtime1++;
         if( rtime1 == 0 )
         {
            rtime0++;
         }
      }
   }
}

void main( void )
{
   rtime3 = rtime2 = rtime1 = rtime0 = 0;
```

File : main.c

Receive Interrupt Time Capture

Now when the CAN message receive interrupt occurs, the current time can be captured just be copying the running time values (rtime) into the captured time values (ctime).

Note that we are only using 24 bits which gives us over 27 minutes of time stamping. If more range is needed, the 4th byte can be used (ctime0 = rtime0).

The online variable is used to turn on and off the 'Receive All CAN Message' feature. This will ultimately be a command from the PC (discussed later) but shown here just so you're aware of this.

```
void interrupt InterruptServiceRoutine( void )
{
   if( RXB0IF )
   {
      if( online == TRUE )
      {
         ecan_get_Rx0();
         // Get the current time, 24 bits only
         ctime1 = rtime1;
         ctime2 = rtime2;
         ctime3 = rtime3;
         send_new_message = TRUE;
      }
      RXB0FUL = 0;
      RXB0IF = 0;
```

File : main.c

CAN Transmit Message Time Queue

Now we'll develop a feature to send multiple CAN messages periodically. Periodic CAN transmit messages will be based off of a 1 millisecond time base and we'll allow up to 8 CAN messages to be configured. Since the 100 microsecond timer interrupt already has been developed, a simple count to 10 strategy is used.

Interrupt : 1 Millisecond Timer

The 100 microsecond interrupt service routine is changed to include a mechanism to provide a 1 millisecond time base.

The UINT8 variable millisecond_timer will count to 10 causing the execution of the SendMessageTimer() function which will trigger the sending of the periodic messages.

```
/* ------------------------------------------------
 * This timer is set to trigger every 100
 * microseconds to provide receive message
 * time stamping.
 * ------------------------------------------------*/
  if( TMR2IF )
  {
    IncrementTime();

    millisecond_timer++;
    if( millisecond_timer >= 10)
    {
      millisecond_timer = 0;
      SendMessageTimer();
    }
    TMR2IF = 0;
  }
```

File : main.c

Structure : Timed Transmit Message

For each periodic message we need a free running counter as well as a value to set the trigger time. The typedef'd structure TIMED_CAN_MESSAGE contains the variable mtime_counter which will count from 0 up to the value contained in mtime which sets the transmit time.

```
typedef struct {
  UINT8 mtime_counter;
  UINT8 mtime;
  CAN_MESSAGE cmess;
} TIMED_CAN_MESSAGE;

#define MAX_TMESS (8)
TIMED_CAN_MESSAGE cm[MAX_TMESS];
```

File : main.c

The structure also contains the CAN_MESSAGE structure which will contain the specific CAN message id, length, and data to send. We then allocate 8 transmit messages in the cm[] array of type TIMED_CAN_MESSAGE. Note that you can increase or decrease the number of transmit messages by changing the MAX_TMESS value.

Function : ClearMessageTimer()

In the initialization section of main() a call to the ClearMessageTimer() function is made to initialize both the counter and limit to 0. The limit is set when the message is configure.

```
void ClearMessageTimer( void )
{
   UINT8 x;

   for( x=0; x< MAX_TMESS; x++ )
   {
      cm[x].mtime_counter = 0;
      cm[x].mtime = 0;
   }
}
```

File : main.c

Function : SendMessageTimer()

The SendMessageTimer() function is called from the interrupt service routine every millisecond (see above). The function processes all defined transmit message by first checking to see if it is timed message. The mtime member of the structure will be non-zero in this case. Next, mtime_counter is incremented then checked against mtime and if the same then the message will be copied to the transmit buffer and sent.

Note that if the transmit buffer is already being transmited then the counter is decremented causing it to be sent on the following millisecond interrupt. Multiple messages could be sent at the same time and this is a clever mechanism to make sure all messages eventually are sent.

```
void SendMessageTimer( void )
{
   UINT8 x;

   for( x=0; x< MAX_TMESS; x++ )
   {
      if( cm[x].mtime > 0 )
      {
         cm[x].mtime_counter++;
         if( cm[x].mtime_counter >= cm[x].mtime )
         {
            // If the tx message buffer is full,
            // then decrement the timer
            // so it will get sent the next time.
            if( (TXB0CON & 0x08) == 0x08 )
            {
               cm[x].mtime_counter--;
            }
            else
            {
               SendTimedMessage( x );
               ecan_send_Tx0();
               cm[x].mtime_counter = 0;
            }
         }
      }
   }
}
```

File : main.c

Note that if any messages are to be sent every millisecond, then any following messages would not get sent. In this case using the other transmit buffers would be necessary.

Function : SendTimedMessage()

The SendTimedMessage() function is called from the SendMessageTimer() to copy the CAN message from the cm[] array to the transmit buffer.

```
void SendTimedMessage( UINT8 mess )
{
   /* set the message length */
   TxMessage0.len    = cm[mess].cmess.len;
   /* set the data values    */
   TxMessage0.dta[0] = cm[mess].cmess.dta[0];
   TxMessage0.dta[1] = cm[mess].cmess.dta[1];
   TxMessage0.dta[2] = cm[mess].cmess.dta[2];
   TxMessage0.dta[3] = cm[mess].cmess.dta[3];
   TxMessage0.dta[4] = cm[mess].cmess.dta[4];
   TxMessage0.dta[5] = cm[mess].cmess.dta[5];
   TxMessage0.dta[6] = cm[mess].cmess.dta[6];
   TxMessage0.dta[7] = cm[mess].cmess.dta[7];
   TxMessage0.id_type = STANDARD;
   TxMessage0.id.sid = cm[mess].cmess.id.sid;
}
```

File : main.c

Chapter 4 - High Speed Serial Interface

Controller Area Networks typically operate at speeds around 500k which means that if we want to upload data to a PC without missing any messages then we need a high speed interface. The USB interface used to communicate with the CanWiser as well as provide power for it can support high-speed communication. We're going to the maximum serial baud rate of 2 Megabaud that the PIC used on the CanWiser can operate at.

Operating at 2 Megabaud will pose 2 issues that will need to be addressed. The first one is that we're not going to be sending Ascii data to be displayed using some terminal program such as Terminal.exe or HyperTerminal. Instead we're going to send formatted binary data from CanWiser to the PC in a data length optimized format. This will be explained later.

The second issue is that although the PC can buffer fast data and process it without any issues, the CanWiser doesn't buffer the incoming data so we'll need to keep this in mind when sending commands from the PC to the CanWiser. This really isn't an big deal since the fast major of data is actually coming off of the CAN bus through the CanWiser to the PC.

Setup : Serial Baud Rate (2 Mega-baud)

The SetupCommPort() function already supports 2 Megabaud by calling the function with the BAUD2000000 value. SetupCommPort() is contained in serial.c and BAUD2000000 is defined in serial.h. The code shown here shows how to setup the baud rate in the initialization section of main().

```
void main( void )
{

   SetupCommPort( BAUD2000000 );

   // other intialization

   while( 1 )
   {
      // background processing
   }
}
```

File : main.c

Communication : CanWiser to the PC

One of the challenges of sending data to the PC on a high-speed interface is that we need a mechanism to sync up the information being passed. A message protocol needs to be developed. In this case we will use a message protocol that uses a binary 0 (0x00) as the start of message identifier as well as use it as the end of message identifier.

The second byte in the message will be a value that points to the next 0 in the message. If there are no 0's in the message then this value will point to the end of the message. If there are 0's in the body of the message (which will happen) then this zero value will be a pointer to the next 0 position. This will guarantee that any 0 seen in the message is either a start or end of message identifier.

The third byte in the message will be a message format byte. The most significant 4 bits of the byte will identify the message and the least significant 4 bits will be used as a data length byte. This results in an efficient variable length message that doesn't waste bytes by adding any place holder values.

Function : SendMessage()

The SendMessage() function formats the rx[] array by adding the start of message byte *rx[0] = 0x00;* and end of message byte *rx[mlen+2]=0x00;* as well as scanning through the message processing the positions with 0 in them into next 0 pointers.

The variable y will point into the rx[] array at the position containing the current zero position count value. It is initialized to 1 meaning the 2nd byte in the message will be the first zero position count.

As the for-next loop scans through the rest of the array, if a zero is found then y is set to this position as the next zero position count.

At the end of the function the message is sent.

```
#define MAX_RX (16)
UINT8 rx[MAX_RX];
UINT8 x,y;

void SendMessage( UINT8 mlen )
{
  rx[0] = 0x00;       // first byte of message always 0
  rx[mlen+2] = 0x00; // last  byte of message always 0
  y = 1;          // point to the first zero count location
  rx[y] = 1;      // init the next zero position
  for( x=2; x < (mlen+2); x++ )
  {
    if( rx[x] != 0 )
    {
      rx[y]++;
    }
    else
    {
      y = x;
      rx[y] = 1;
    }
  }
  CommPortSendByte( 0 );
  for( x=1; x <= (mlen+2); x++)
     CommPortSendByte( rx[x] );
}
```

File : main.c

Function : SendCANdata2PC()

The SendCANdata2PC() function copies the received CAN message into the rx[] array in the proper format.

```
UINT8 len, lx, ly;
void SendCANdata2PC( void )
{
  ly = 0;
  len = rx[2] = RxMessage0.len;
  if( send_time == TRUE )
  {
    rx[2] |= 0x10;
    ly = 3;
    rx[3] = ctime1;
    rx[4] = ctime2;
    rx[5] = ctime3;
  }
  rx[3+ly] = (*(((unsigned char *)&RxMessage0.id.sid)+1));
  rx[4+ly] = (*(((unsigned char *)&RxMessage0.id.sid)+0));
  for( lx=0; lx<len; lx++ )
  {
     rx[5+lx+ly] = RxMessage0.dta[lx];
  }
  SendMessage( 3+len+ly );
}
```

File : main.c

Feature : Sending All Received Messages

In the main() function a check of the send_new_message flag is made and if TRUE then the received message is sent to the PC using the SendCANdata2PC() function.

```
void main( void )
{
   // initialization stuff goes here

   while( 1 )
   {
      if( send_new_message == TRUE )
      {
          SendCANdata2PC();
           send_new_message = FALSE;
      }
   }
```

File : main.c

Communication : PC to the CanWiser

Sending data from the PC to the CanWiser will consist of a fixed length message with 2 start of message identifier bytes. Although the PIC used on the CanWiser is fast, the PC can send data to it fast enough where it won't be able to keep up. Therefore an efficient message format isn't really necessary and there isn't a lot of data that needs to be sent to the CanWiser anyways.

Interrupt : Serial Receive

The interrupt service routine needs to check the RCIF flag to see if a new character has arrived. If it has, the flag is reset, the character is stored, and then processed on the ProcessIncoming() function.

Note the the serial receive interrupt (bit 5) in register PIE1 must be set to 1 (shown in main()).

```
void interrupt InterruptServiceRoutine( void )
{
    // Check for a serial receive interrupt
  if( RCIF == 1 )
  {
    /* clear the serial receive interupt flag */
    RCIF = 0;
    /* read the received character            */
    rxin = RCREG;
    ProcessIncoming();
  }
}

void main( void )
{
/* Bit 5 RCIE=1 Enable serial receive interrupt */
PIE1 = 0b00100010;
```

File : main.c

Setup : PC Received Message Buffer & State Machine

The 2 start characters are defined at 0xaa and 0x55 respectively and can actually be set to whatever is needed. The variable cmmd_state will contain the state of the received message and can take on 1 of 3 states.

In the first and second states the 2 start bytes are being looked for. Once these are found, then the fixed length data will be stored.

```
#define START1_CHARACTER  0xaa
#define START2_CHARACTER  0x55

#define WAIT4_START1   0
#define WAIT4_START2   1
#define WAIT4_MESSAGE  2
UINT8 cmmd_state;

UINT8 rxin_count;
#define MAX_RXIN  10
UINT8 rxin_array[MAX_RXIN];
```

File : main.c

Function : ProcessIncoming()

The ProcessIncoming() function is the state manager for identifying the message.

Initially the cmmd_state is looking the 1st start character. When this is sent and identified, the state is changed to look for the second start byte. After both correct start bytes are received, the fixed length data is looked for.

Once a complete message is received, the ProcessCommand() function is used to decode the message.

```
void ProcessIncoming( void )
{
   switch( cmmd_state )
   {
      case WAIT4_START1:
           if( rxin == START1_CHARACTER )
           {
                cmmd_state = WAIT4_START2;
           }
           break;
      case WAIT4_START2:
           if( rxin == START2_CHARACTER )
           {
                cmmd_state = WAIT4_MESSAGE;
                rxin_count = 0;
           }
           break;
      case WAIT4_MESSAGE:
           rxin_array[rxin_count] = rxin;
           rxin_count++;
           if( rxin_count >= MAX_RXIN )
           {
             if( rxin == 0x00 )
             {
                 ProcessCommand();
                 cmmd_state = WAIT4_START1;
             }
           }
           break;
   }
}
```

File : main.c

Function : ProcessCommand()

The ProcessCommand() function decodes the incoming serial message.

```
#define OFFLINE_COMMAND    (0x20)
#define ONLINE_COMMAND     (0x21)
#define SETUP_MESSAGE0     (0)
UINT8 pcx;
void ProcessCommand( void )
{
   if( rxin_array[0] < MAX_TMESS )
   {
      if( (rxin_array[1] & 0xf0) == 0 )
      {
        cm[rxin_array[0]].cmess.id_type = STANDARD;
      }
      else
      {
        cm[rxin_array[0]].cmess.id_type = EXTENDED;
      }
      cm[rxin_array[0]].cmess.len = rxin_array[1] & 0x0f;
      cm[rxin_array[0]].cmess.id.sid =
                    (rxin_array[2] << 8) + rxin_array[3];
      cm[rxin_array[0]].mtime = rxin_array[4];
   }
   else
   {
      if( (rxin_array[0] & 0xf0) == 0x10 )
      {
        pcx = rxin_array[0] & 0x0f;
       if( pcx < MAX_TMESS)
       {
          cm[pcx].cmess.dta[0] = rxin_array[1];
          cm[pcx].cmess.dta[1] = rxin_array[2];
          cm[pcx].cmess.dta[2] = rxin_array[3];
          cm[pcx].cmess.dta[3] = rxin_array[4];
          cm[pcx].cmess.dta[4] = rxin_array[5];
          cm[pcx].cmess.dta[5] = rxin_array[6];
          cm[pcx].cmess.dta[6] = rxin_array[7];
          cm[pcx].cmess.dta[7] = rxin_array[8];
       }
      }
      else
      {
        switch( rxin_array[0] )
        {
          case OFFLINE_COMMAND: online = FALSE; break;
          case ONLINE_COMMAND:  online = TRUE; break;
        }
      }
   }
}
File : main.c
```

Commands

The command byte identifies what the CanWiser is to do with the message and is detailed in the following table.

Command Byte Value(s)	***Message Name***	***Description***
0x00 to MAX_TMESS	Setup Transmit Message	The command byte indicates which transmit message to configure. The next contains both the CAN id type and the message length. If bit4 is 0 then the ID is 11 bits otherwise it is the 29 bit identifier. The 4 low order bits are the message length.
0x10 to (0x10 + MAX_TMESS)	Set Transmit Message Data Values	
0x20	Receive Offline	Receiving all CAN messages is disabled
0x21	Receive Online	Receiving all CAN messages is enabled
0x22	Version	Send the CanWiser version

Chapter 5 - The PC Program

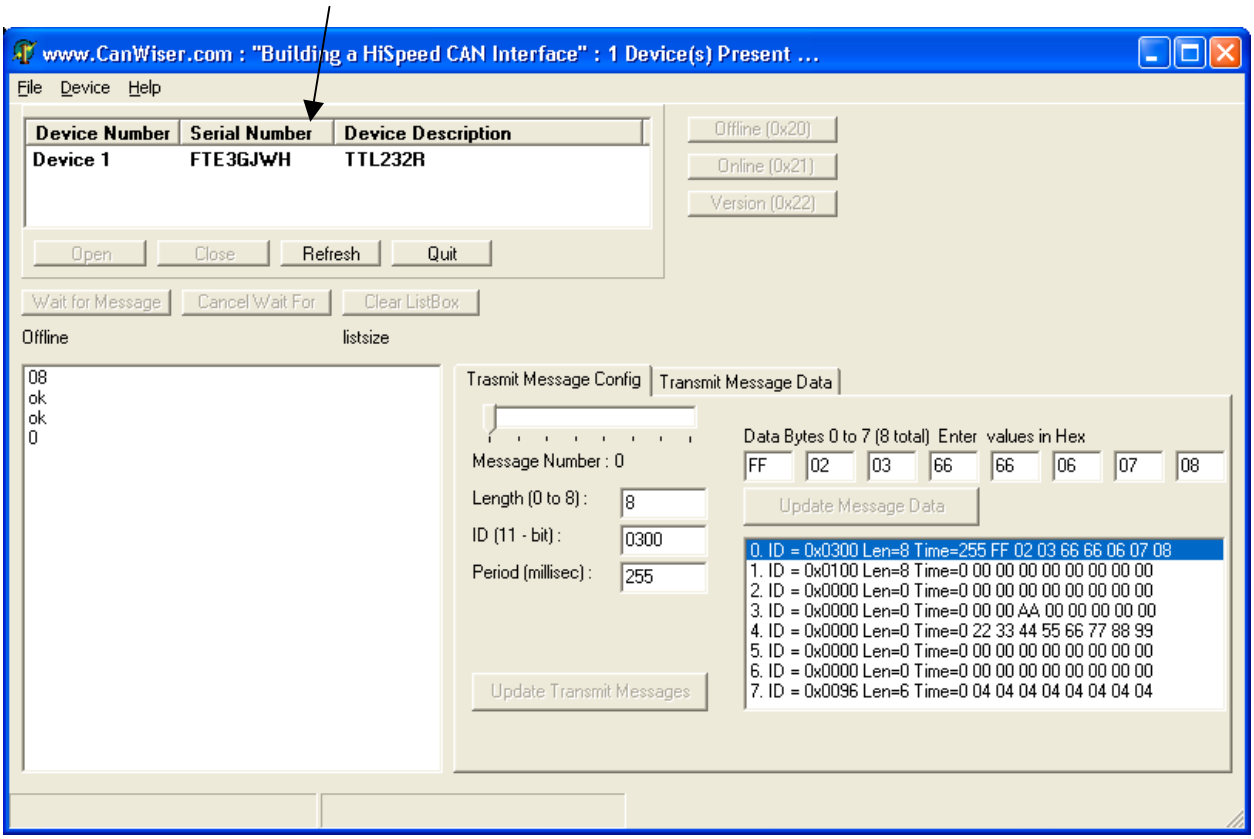
The PC interface program will use the Borland Delphi programming language to talk to the CanWiser. Delphi is based on the Pascal language and is a straightforward PC programming language that is easy to use if you know 'C'. The source code as well as the executable is available at www.emicros.com/canwiser_interface.zip.

Running the CanWiser PC Program

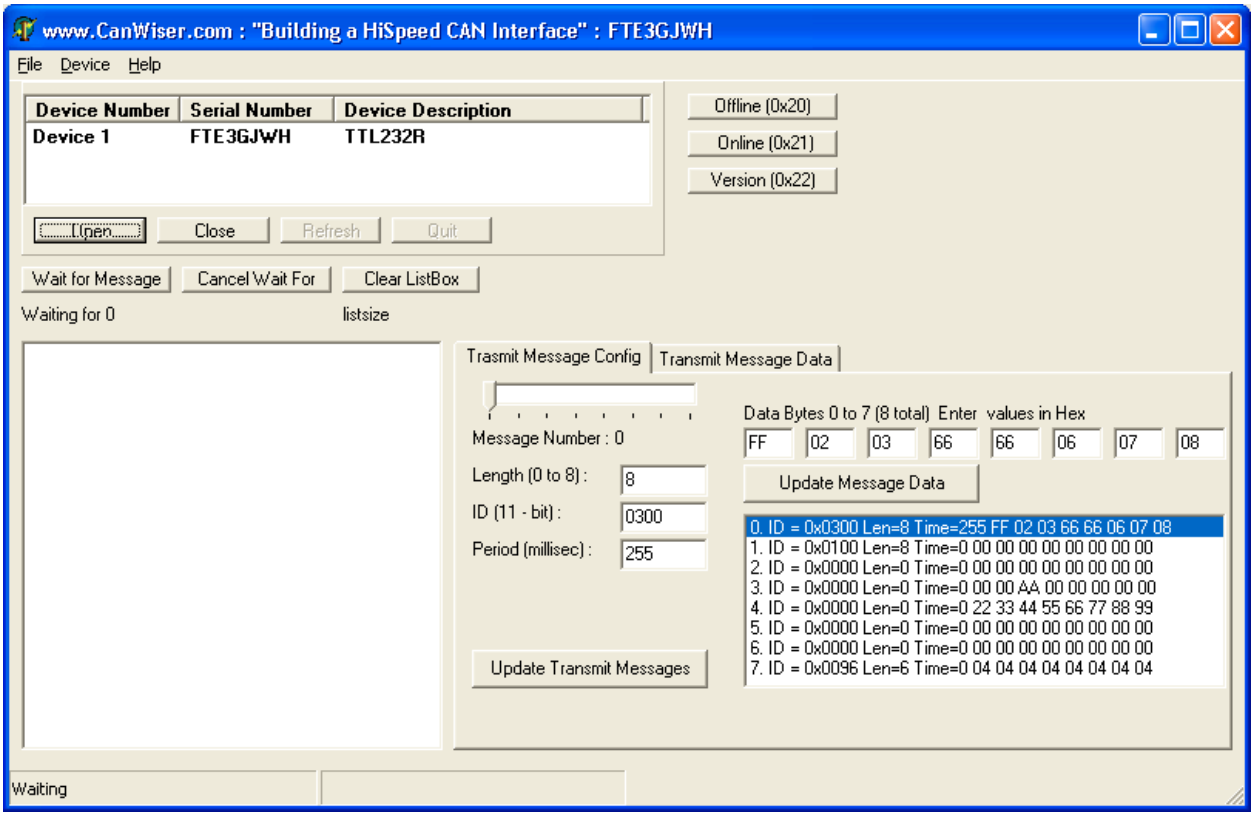
Unzip the canwiser_interface.zip file into a directory such as c:\CanWiser_if and then execute the CanWiser.exe file. You will get a window similar to the one shown in the following section.

Finding the CanWiser Interface

The first thing the program does is to search if any device are present. In this case a CanWiser with serial number FTE3GJWH is identified.



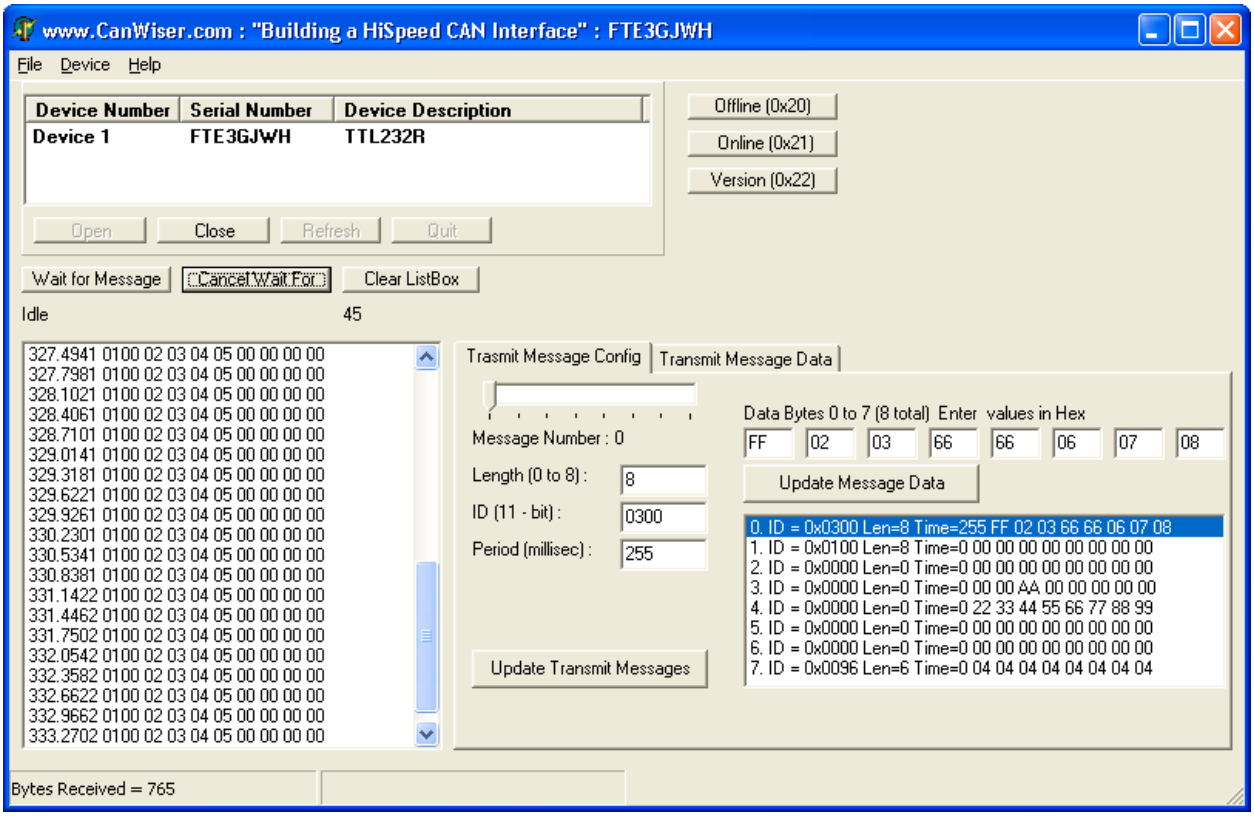
Click anywhere on the line "Device 1 … TTL232R" to highlight if and then press the Open button. Once CanWiser is opened the 'buttons' are enabled and you're ready to send and receive CAN messages.



The CanWiser is now open and ready for use.

Receiving All CAN Message Traffic

Click on the "Online (0x21)" button and then the "Wait for Message" button. If you are connected to a CAN network and there is message traffic you will see the message receive time, identifier, and data bytes in the list box.

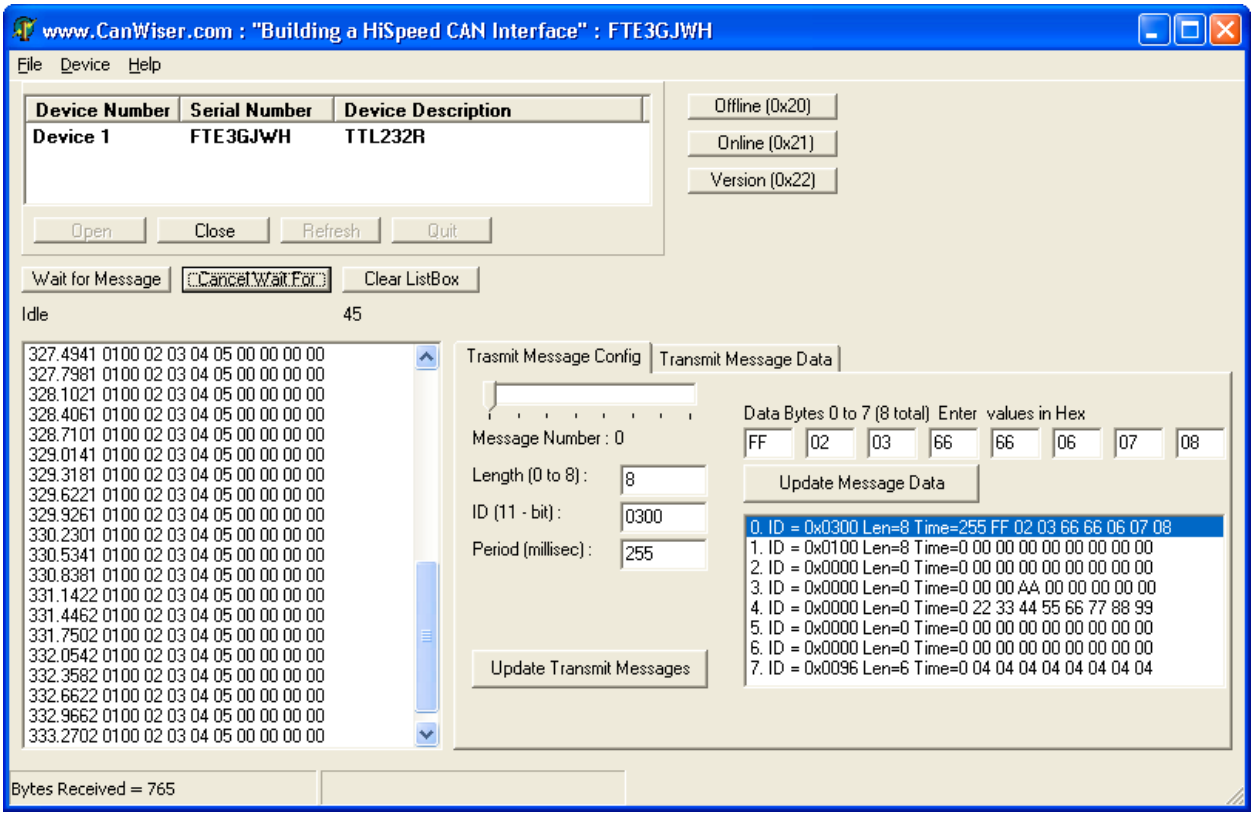


Sending Periodic CAN Messages

In order to have CanWiser send CAN messages periodically, use the Transmit Message Config tab to select the message number, ID, Length, and Period. Hit the "Update Transmit Messages" button to update the transmit messages in CanWiser.

Setting the data in the message requires the Message Number to be selected (using the slider bar) and then entering it as Hex values. Hit the "Update Message Data" button to send the data to CanWiser.

Only 8 Transmit Messages are allowed and this is the default setting in the CanWiser application.



Sending Single CAN Messages

You can send a single CAN message by setting the period for that message to 255.