Development Kit For the PIC® MCU

Exercise Book

CAN Bus 24

March 2010

Custom Computer Services, Inc.
Brookfield, Wisconsin, USA
262-522-6500

Recognized Microchip Third-Party Tool Provider

Custom Computer Services, Inc. proudly supports the Microchip brand with highly optimized C compilers and embedded software development tools.

1 UNPACKING AND INSTALLATION

Inventory

- ❑ Use of this kit requires a PC with Windows 95, 98, ME, NT, 2000 or XP. The PC must have a spare 9-Pin Serial or USB port, a CD-ROM drive and 75 MB of disk space.
- ❑ The diagram on the following page shows each component in the CAN Bus 24 kit. Ensure every item is present.

Software

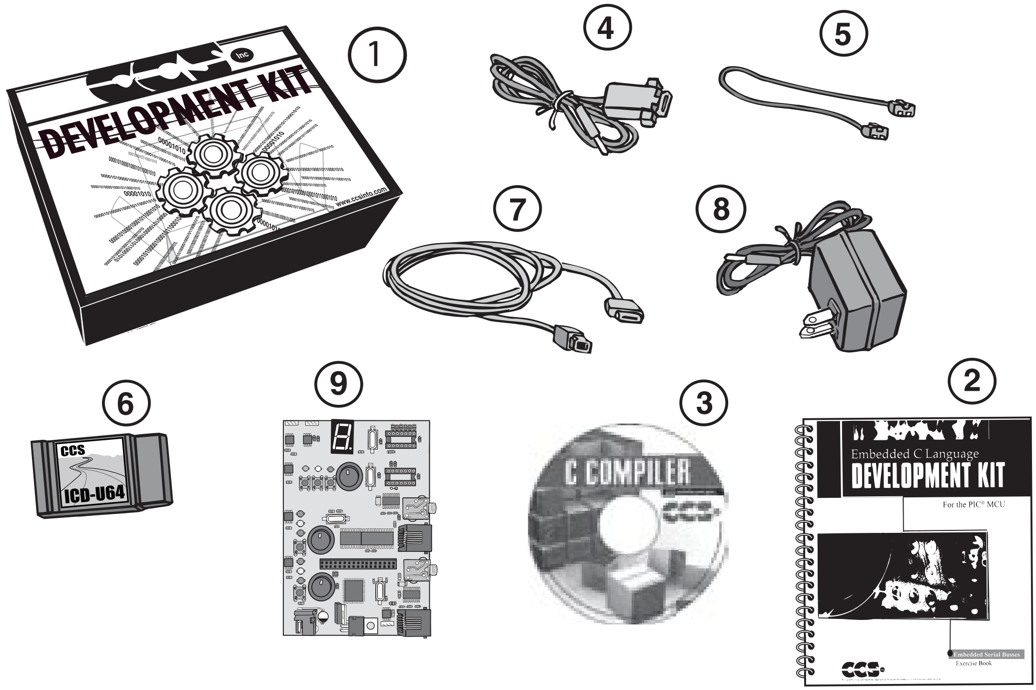
- ❑ Insert the CD into the computer and wait for the installation program to start. If your computer is not set up to auto-run CDs, then select **My Computer** and double-click on the CD drive.
- ❑ Click on **Install** and use the default settings for all subsequent prompts by clicking NEXT, OK, CONTINUE…as required.
- ❑ Identify a directory to be used for the programs in this booklet. The install program will have created an empty directory **c:\program files\picc\projects** that may be used for this purpose. However, the example files can be found on the compiler CD-ROM, located in the directory **D:/CCS\CAN Bus 24 Exercise Book.**
- ❑ Select the compiler icon on the desktop. In the PCW IDE, click **Help>About** and verify a version number is shown for the IDE and PCD to ensure the software was installed properly. Exit the software.

Hardware

- ❑ Connect the PC to the ICD(4) using the USB cable.(1) Connect the prototyping board (8) to the ICD using the modular cable. Plug in the DC adaptor (7) to the power socket and plug it into the prototyping board (8). The first time the ICD-U is connected to the PC, Windows will detect new hardware. Install the USB driver from the CD or website using the new hardware wizard. The driver needs to be installed properly before the device can be used.
- ❑ The LED should be red(2) on the ICD-U to indicate the unit is connected properly.
- ❑ Run the Programmer Control Software by clicking on the CCSLOAD icon on the desktop. Use CCSLOAD Help File for assistance.
- ❑ The software will auto-detect the programmer and target board and the LED should be illuminated green. If any errors are detected, go to Diagnostic tab. If all tests pass, the hardware is installed properly.
- ❑ Disconnect the hardware until you are ready for Chapter 3. Always disconnect the power to the Prototyping board before connecting/disconnecting the ICD or changing the jumper wires to the Prototyping board.

(1) ICS-S40 can also be used in place of ICD-U. Connect it to an available serial port on the PC using the 9 pin serial cable. There is no driver required for S40.

(2) ICD-U40 units will be dimly illuminated green and may blink while connecting.



1. Storage box
2. Exercise booklet
3. CD-ROM of C compiler (optional)
4. Two Serial PC to Prototyping Board Cables
5. Modular ICD to Prototyping board cable
6. ICD unit for programming and debugging
7. USB (or Serial) PC to Prototyping board cable
8. AC Adaptor (9VDC)
9. CAN Bus 24 Prototyping board

2 USING THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Editor

- Open the PCW IDE. If any files are open, click **File>Close All**
- Click **File>Open>Source File**. Select the file: **c:\program files\picc\examples\ex_stwt.c**
- Scroll down to the bottom of this file. Notice the editor shows comments, preprocessor directives and C keywords in different colors.
- Move the cursor over the **Set_timer0** and click. Press the F1 key. Notice a Help file description for set_timer0 appears. The cursor may be placed on any keyword or built-in function and F1 will find help for the item.
- Review the editor special functions by clicking on **Edit**. The IDE allows various standard cut, paste and copy functions.
- Review the editor option settings by clicking on **Options>Editor Properties**. The IDE allows selection of the tab size, editor colors, fonts, and many more. Click on **Options>Toolbar** to select which icons appear on the toolbars.

Compiler

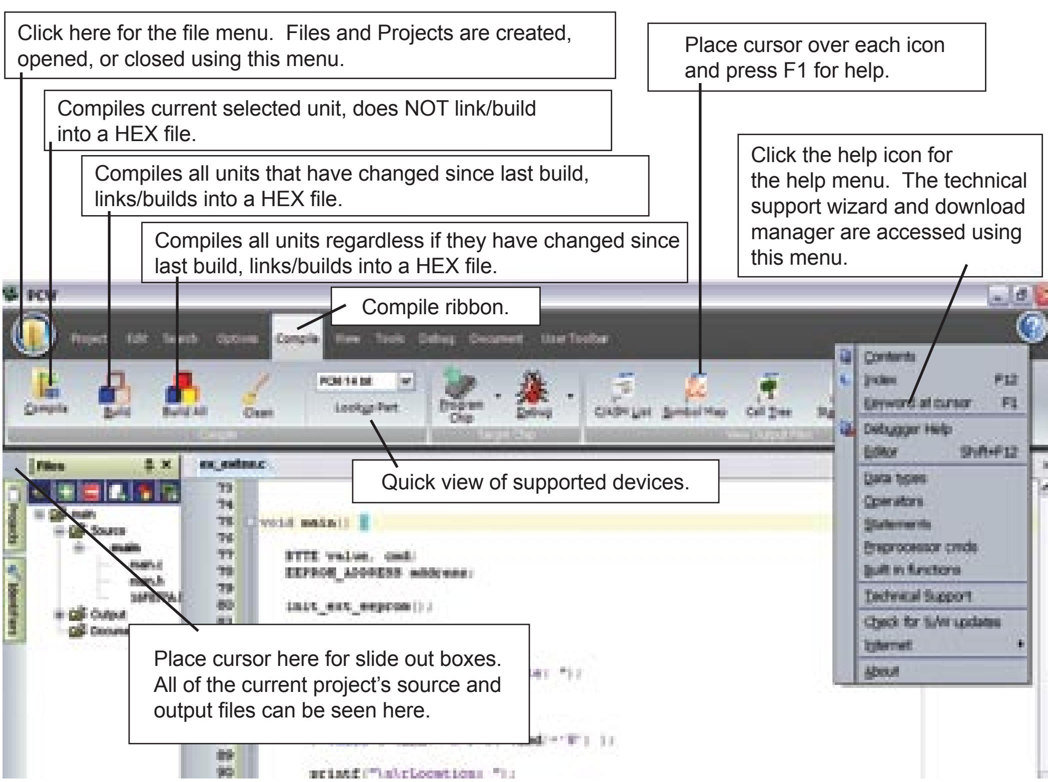
- Use the drop-down box under Compile to select the compiler. CCS offers different compilers for each family of Microchip parts. The exercises in this booklet are for the PIC24HJ256GP610 and dsPIC30F4012 chips, 24-bit opcode parts.. Make sure **PCD 24 bit** is selected in the drop-down box under the **Compiler** tab.
- The main program compiled is always shown in the bottom of the IDE. If this is not the file you want to compile, then click on the tab of the file you want to compile. Right click into editor and select **Make file project**.
- Click **Options>Project Options>Output Files…** and review the list of directories the compiler uses to search for included files. The install program should have put two directories in this list to point to the device: *.h files and the device drivers*.
- Normally the file formats need not be changed and global defines are not used in these exercises. To review these settings, click **Options>Project Options>Output Files** and **Options>Project Options>Global Defines**.
- Click the compile icon to compile. Notice the compilation box shows the files created and the amount of ROM and RAM used by this program. Press any key to remove the compilation box.

Viewer

- ❑ Click **Compile>Symbol Map**. This file shows how the RAM in the micro-controller is used. Identifiers that start with @ are compiler generated variables. Notice some locations are used by more than one item. This is because those variables are not active at the same time.
- ❑ Click **Compile>C/ASM List.** This file shows the original C code and the assembly code generated for the C. Scroll down to the line:

```
int_count=INTS_PER_SECOND;
```

- ❑ Notice there are two assembly instructions generated. The first loads 4C into the W register. INTS_PER_SECOND is #defined in the file to 76. 4C hex is 76 decimal. The second instruction moves W into memory. Switch to the Symbol Map to find the memory location where int_count is located.
- ❑ Click **View>Data Sheet**, then **View.** This brings up the Microchip data sheet for the microprocessor being used in the current project.



3 CAN BUS PROTOTYPING BOARD OVERVIEW

- The CCS CAN Bus 24 prototyping board has a CAN bus with four nodes on the same board. A block diagram is within the front cover of this booklet. The four independent nodes are as follows:

NODE A - PIC24HJ256GP610

- This node features a Microchip PIC24HJ256GP610 chip. This chip has two built-in ECAN bus controllers. There is also an I/O block that provides access to spare I/O pins on the PIC. The pinout is as follows:

+3.3	E4	E2	G14	G12	E0	A6	D6	D4	D12	D2	D0	C14	D10	D8	A14	G
+3.3	E5	E3	G15	G13	E1	A7	D7	D5	D13	D3	D1	C13	D11	D9	A15	G

The following I/O features are also a part of NODE A:

- Three LEDs (Red, Yellow, Green)
  LED is lit by outputting a LOW to the I/O pin
- One push-button
  I/O pin reads LOW when the button is pressed
- Pot to provide an analog voltage source
  0 Volts full counterclockwise, 3.3 Volts full clockwise
- RS-232 port

NODE B - dsPIC30F4012

- This node features a Microchip dsPIC30F4012 chip which is connected to an external MCP2515 CAN bus controller. This scheme could be used with any PIC microcontroller.

The following I/O features are also a part of NODE B:

- Three LEDs (Red, Yellow, Green)
  LED is lit by outputting a LOW to the I/O pin
- One push-button
  I/O pin reads LOW when the button is pressed
- Pot to provide an analog voltage source
  0 Volts full counterclockwise, 5 Volts full clockwise
- RS-232 port

Programs may be downloaded and optionally debugged using the ICD connector.

NODE C - "Dumb" I/O Unit

- ❑ This node features a Microchip MCP25050 chip. This chip is pre-programmed with address information and provides CAN bus access to the eight I/O pins. The following items are connected to the I/O pins:
  - Three LEDs (Red, Yellow, Green)
    LED is lit by outputting a LOW to the I/O pin
  - Three push-buttons
    I/O pin reads LOW when the button is pressed
  - Pot to provide an analog voltage source
    0 Volts full counterclockwise, 5 Volts full clockwise
  - RS-232 port

  This chip may be programmed by removing it from the socket and using the Pro Mate II from Microchip to load in the address information.

NODE D - "Dumb" 7 Segement LED

- ❑ This node features a Microchip MCP25050 chip. This chip is pre-programmed with address information and provides CAN bus access to the eight I/O pins. The I/O pins are connected to a 7-segment LED. This allows a number to be displayed via the CAN bus. A LOW on the I/O pin lights the segment. For example, outputting a 0xC0 in the GP port will light a 0. A 0xF9 will light a 1.

  This chip may be programmed by removing it from the socket and using the Pro Mate® II from Microchip to load in the address information.

NOTES

Both Node C & D use a Microchip MCP25050 CAN Bus chip.

- This chip is a complete CAN Bus Node that allows eight general input or output pins, up to four A/D converter inputs and two PWM outputs
- This chip can be configured by programming an internal EEPROM with the addresses and modes of operation.
- The chip can also be programmed over the CAN Bus.

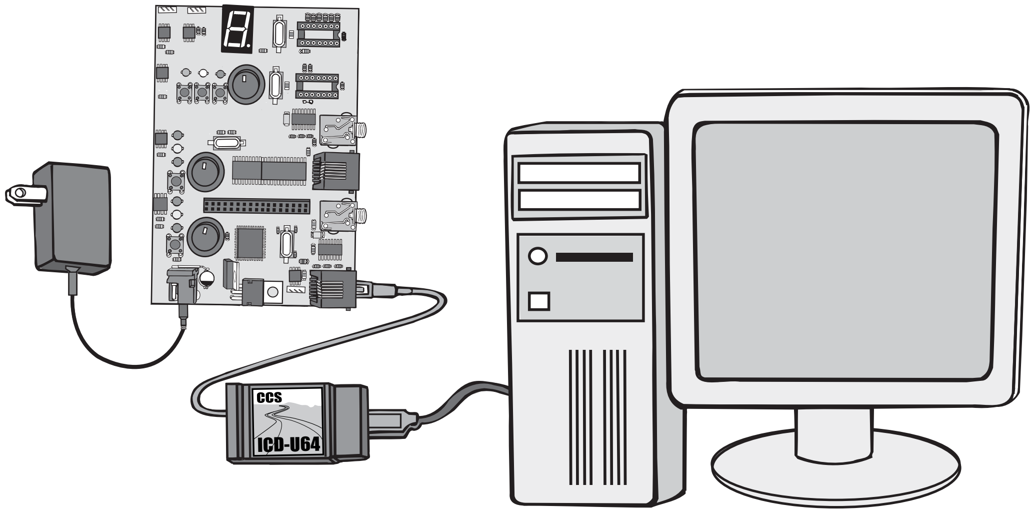
4 COMPILING AND RUNNING A PROGRAM

- Open the PCW IDE. If any files are open, click **File>Close All**
- Click **File>New>Source File** and enter the filename **EX4.C**
- Type in the following program and **Compile.**

```
#include <24HJ256GP610.h>
#device ICD=TRUE
#fuses HS,NOLVP,NOWDT,
#use delay(clock=20M)

#define GREEN_LED PIN_A5

main ()
{
   while(TRUE)
   {
      output_low(GREEN_LED);
      delay_ms(1000);
      output_high(GREEN_LED);
      delay_ms(1000);
   }
}
```

NOTES

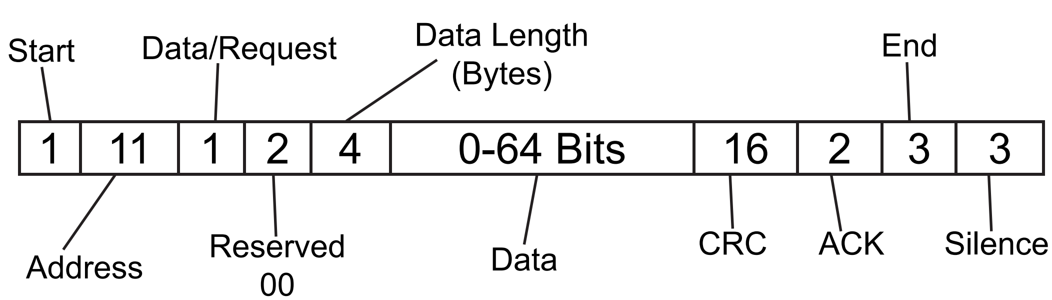
- The first four lines of this program define the basic hardware environment. The chip being used is the PIC24HJ256GP610, running at 20MHz with the ICD debugger.
- The #define is used to enhance readability by referring to GREEN_LED in the program instead of PIN_A5.
- The “while (TRUE)” is a simple way to create a loop that never stops.
- Note that the “output_low” turns the LED on because the other end of the LED is +3.3V. This is done because the chip can tolerate more current when a pin is low than when it is high.
- The “delay_ms(1000)” is a one second delay (1000 milliseconds).



- ❑ Connect the ICD to the Prototyping board using the modular cable, and connect the ICD to the PC. Power up the Prototyping board.
- ❑ Click **Debug>Enable Debugger** and wait for the program to load.
- ❑ If you are using the ICD-U and the debugger cannot communicate to the ICD unit go to the debug configure tab and make sure ICD-USB from the list box is selected.
- ❑ Click the green go icon: 
- ❑ Expect the debugger window status block to turn yellow indicating the program is running.
- ❑ The green LED on the Prototyping board should be flashing. One second on and one second off.
- ❑ The program can be stopped by clicking on the stop icon: 

5 CAN BUS OVERVIEW

- **SIMPLE 4 NODE EXAMPLE:**
  - Node A: Speed detector
    Every 100ms sends a frame with identifier 1 and data indicating the vehicle speed.
  - Node B: Speedometer display
    Looks only for identifier 1 data on the bus. Takes the data and displays it on a digital display.
  - Node C: Cruise control panel
    Pressing the SET button sends an identifier 2 frame.
    Pressing the OFF button sends an identifier 3 frame.
    Neither frame has data.
  - Node D: Cruise control module
    Module turns on with an identifier 2 frame and off with an identifier 3 frame. The module uses data from identifier 1 frames to adjust the vehicle speed.
- This is not a command/response type of protocol. Nodes that have something to say will say it. Nodes that need to know something will wait for what is needed. A higher level protocol can be implemented to provide more control. Node C can actually control how Node D works. If a node needs a certain type of data, it can post a request on the bus for a frame with a particular identifier. The node responsible for that identifier will respond. A system design should assign a given identifier (or set of identifiers) to only one node.
- **BASIC FRAME FORMAT:**
- See Chapter 6 for the Extended Format.



- **GENERAL RULES:**
  - All nodes on the bus verify the frame. If any node detects an error, that node asserts a NACK. When any node asserts a NACK for a frame, all nodes must ignore the frame, even if the node did not find an error in the frame. The sender re-transmits NACKed frames.
  - A node that NACKs a lot of messages or has a lot of messages NACKed is put on probation (Error Passive state). In this state, the node's activity is restricted. If the problem persists, the node must stop all bus transmission and ignore all incoming packets. This rule is self-enforced by each node keeping local statistics.
  - A node does not start transmitting unless the bus is quiet for three bit times. If two nodes start a frame at the same time, one node will bow out while the identifier is being transmitted. The node to drop out will be the one that first tries to send a 1-bit, when the other sends a 0. The 0 is dominant and the sender of the 1 will realize there is a collision. This means lower numbered identifiers have a higher priority.
  - The CAN bus permits an alternate format message with a 29 bit identifier. All the examples we use will be with an 29 bit identifier. Frames with 11 and 29 bit identifiers can co-exist on the same bus.
- **PHYSICAL:**
  - There is no universal standard for the physical CAN Bus. It requires an open drain type of bus. It could be a single wire, fiber optic or two wire differential bus. The latter is the physical bus used on the CAN Bus 24 Prototyping board. The Philips PCA82C251 chips are used to interface the bus to the TTL controllers. This complies with ISO standard 11898 for use in Automotive and Industrial applications
  - The bit rate can be as fast as one million bits per second.
  - The start of frame bit is used by the receiver to determine the exact bit time.
  - Whenever a transmitter on the bus sends five identical bits, it will send an extra bit with the reverse polarity. This is referred to as a stuffed bit. The receiver will ignore the stuffed bits. If a receiver detects six or more bits that are the same, it is considered an automatic error.

6 SIMPLE PIC18 TRANSMITTER

- Click **File> New> Source File** and enter the filename **EX6.c**.Type in the following program:

```
#include <24HJ256GP610.h>
#fuses HS,NOWDT
#use delay(clock=20M)

#define CAN_BRG_PRESCALAR              4
#define CAN_BRG_PHASE_SEGMENT_1        2
#define CAN_BRG_PHASE_SEGMENT_2        2
#define CAN_BRG_SYNCH_JUMP_WIDTH       0
#define CAN_BRG_PROPAGATION_TIME       0

#include <can-PIC24.c>

#define WRITE_REGISTER_D_ID 0x400

void write_7_segment(int8 value){
   const int8 lcd_seg[10]={ 0x40,0x79,0x24,0x30,0x19,
                            0x12,0x02,0x78,0x00,0x10};
int8 buffer[3];

buffer[0]=0x1E;                   //addr of gplat
buffer[1]=0x7F;                   //mask
buffer[2]=lcd_seg[value];
can_putd(WRITE_REGISTER_D_ID, buffer, 3, 1, TRUE, FALSE);
}

void main()
{
   int i=0;

   can_init();                         //initializes can

   can_enable_b_transfer(TRB0);        //enables buffer 0 in transmit mode

   can_putd(0x100,0,0,1,TRUE,FALSE);   //send an on-bus message
                                       //to wake up mcp250x0's
   delay_ms(1000);                     //wait for node d to power-up
   while(TRUE)
   {
      write_7_segment(i);
      delay_ms(1000);
      if( ++i==10)
         i=0;
   }
}
```

- [ ] **Compile** the program. Load into Node A, and run the program.
- [ ] This program should display 0-9 on the 7-segment LED.

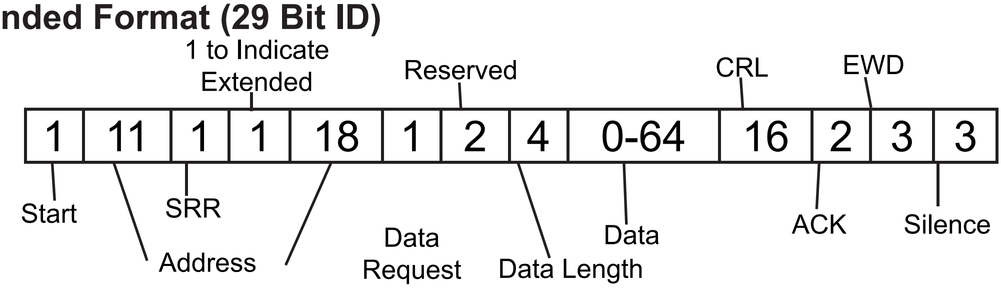
NOTES

- The #define CAN_BRG_PRESCALAR, #define CAN_BRG_PHASE_SEGMENT_1, #define CAN_BRG_PHASE_SEGMENT_2, #define CAN_BRG_SYNCH_JUMP_WIDTH, and #define CAN_BRG_PROPAGATION_TIME are used to setup the PIC24HJ256GP610 ECAN module to a bit time of 125000 bits per second.
- The include file "can-PIC24.c" has the functions required to interface to the PIC24 ECAN Bus controller.
- The call to can_init() starts the interface.
- The call to can_enable_b_transfer enables buffer 0 as a transmit buffer.
- This program is designed to send data to Node D. The identifier for Node D is programmed as 0x400. The MCP25050 device accepts a three byte command.
- The can_putd functions have the following parameters:
  - Identifier
  - Data pointer
  - Number of data bytes
  - Priority (0-3) determines the order messages are sent
  - Flag to indicate 29 bit identifier
  - Flag to indicate if this is a data frame (FALSE) or request for frame (TRUE)
- This call query up a frame for transmission on the bus.
- The MCP250xx units require one error-free message after power-up to switch to normal state. The first CAN_putd, to 0x100, sends an empty message which takes the MCP250xx from power-up to normal.

Before Moving On:

- [ ] Copy the lines in this example before "void main() {" into an include file named CCSCANA.C. Future examples will add to this file and build a library of functions specific to the CAN Bus 24 Prototyping board.

Extended Format (29 Bit ID)



7 DEBUGGING

- ❑ Open the code for chapter 6, add #device ICD =TRUE, and start the debugger **Debug>Enable Debugger**.
- ❑ Click the reset icon to ensure the target is ready.
- ❑ Click the step-over icon . This is the step over command. Each click causes a line of C code to be executed. The highlighted line has not been executed, but the line about to be executed.
- ❑ Step over the `can _ init;` line and notice that one click executed the entire function. This is the way step over works. Click step over on `delay=ms(1000);`. and notice the debugger stops when the function terminates.
- ❑ Click the **Watch** tab, then the add icon to add a watch. Enter **i or choose i the variables from list**, then click **Add Watch**. Notice the value shown. Continue to step over through the loop a few more times and notice the count watch increments.
- ❑ Step over until the call to `write _ 7 _ segment(i);` is highlighted. This time, instead of step over, use the standard step icon several times and notice the debugger is now stepping into the function.
- ❑ Click the GO icon to allow the program to run. Click the stop icon to halt execution. Notice the C source line that the program stopped on.
- ❑ In the editor, click on `write _ 7 _ segment(i);` to move the editor cursor to that line. Then click the Breaks tab and click the add icon to set a breakpoint. The debugger will now stop every time that line is reached in the code. Click the GO icon. The debugger should now stop on the breakpoint. Repeat this a couple of times to see how the breakpoint works.
- ❑ Click **Compile>C/ASM list**. Scroll down to the highlighted line. Notice that one assembly instruction was already executed for the next line. This is another side effect of the ICD-S debugger. Sometimes breakpoints slip by one ASM instruction.
- ❑ Click the step over icon a few times and note that when the list file is the selected window, the debugger has executed one assembly instruction per click instead of one entire C line.
- ❑ Close all files and start a new file **EX7.C** as follows:

```
#include <24J250GP610.h>
#device ICD=TRUE

void main() {
    int a,b,c;

    a=11;
    b=5;
    c=a+b;
    c=b-a;
    while(TRUE);
}
```

- ❑ **Compile** the program and step-over until the c=a+b is executed. Add a watch for c and the expected value is 16.
- ❑ Step-over the subtraction and notice the value of c. The **int** data type by default is not signed, so c cannot be the expected –6. The modular arithmetic works like a car odometer when the car is in reverse only in binary. For example, 00000001 minus 1 is 00000000, subtract another 1 and you get 11111111.
- ❑ Reset and again step up to the c=a+b. Click the **Eval** tab. This pane allows a one time expression evaluation. Type in a+b and click **Eval** to see the debugger and calculate the result. The complete expression may also be put in the watch pane as well. Now enter b=10 and click **Eval**. This expression will actually change the value of B if the "keep side effects" check box of the evaluation tab is checked. Check it and click **Eval** again. Step over the addition line and click the **Watch** tab to observe the c value was calculated with the new value of b.

FURTHER STUDY

A *Modify the program to include the following C operators to see how they work:*
** / % & ^*
Then, with b=2 try these operators: >> <<
Finally, try the unary complement operator with: c=~a;

B *Design a program to test the results of the relational operators:*
< > = = !=
by exercising them with b as 10, 11, and 12.
Then, try the logical operators || and && with the four combinations of a=0,1 and b=0,1.
Finally, try the unary not operator with: c=!a; when a is 0 and 1.

8 USING THE MCP250XX FOR OUTPUT

- The MCP250xx parts used on Nodes C and D allow for discrete input, output and analog input. These parts have internal registers that set the device ID, the directions of the pins, values of the outputs, scheduling information for outgoing frames, and more. These registers are initialized by programming the part on a Microchip Pro Mate II. The registers can also be read and modified at run time.
- The MCP250xx part for Node D has been programmed with a base ID of 0x400. The low three bits of the ID specify a function. For example, 0x400 is a write-register command and 0x404 is a write-configuration command. Table 4-2 in the data sheet explains the identifier usage.
- The write-register command has three bytes of data namely, a register, mask, and value. The value is written to the specified register changing only the bits specified in the mask. For example, in the previous program, a frame was sent with ID 0x400 and data 0x1E, 0x7F, 0x40. 0x1E is the output latch for the GP pins. 0x7F caused GP7 to be unchanged (connected to decimal point). The value 0x40 puts a low on pins GP0 to GP5 and a high on GP6. Note that the registers listed in the data sheet table 3-1 use addresses for the internal EEPROM. The RAM addresses are 0x1C higher.
- Example:
  Send a frame with ID 0x400 and data 0x1E, 0x80, 0x00 to turn on the DP

  0x400 -- Node D
  0x1E -- Output Latch register
  0x80 -- Only change Bit 7
  0x00 -- All zeros (only Bit 7 relevant) A 0 or low lights the segment
- Compile and Run the example. Verify that the DP is on.
- Node C has three LEDs: Red (GP1), Yellow (GP2) and Green (GP3). Add the following function to CCSCANA.C:

```
#define WRITE_REGISTER_C_ID 0x300

enum colors {RED=0,YELLOW=1,GREEN=2};

void write_c_led(colors led, short on) {
   int8 buffer[3];

   buffer[0]=0x1E;
   buffer[1]=0x02<<led;

   if(on)
      buffer[2]=0;
   else
      buffer[2]=0xff;
   can_putd(WRITE_REGISTER_C_ID, buffer, 3, 1, TRUE, FALSE); }
```

- [ ] Click **File> New > Source File>** and enter filename **EX8A.C**
- [ ] Type in the following program:

```
#include "CCSCANA.c"

void main()
{
   int i=0;

   can_init();              // always initialize the can

   can_enable_b_transfer(TRB0);   //enable buffer 0 as transmit buffer
   can_enable_b_transfer(TRB1);   //enable buffer 1 as transmit buffer
   can_enable_b_transfer(TRB2);   //enable buffer 2 as transmit buffer

   can_putd(0x100,0,0,1,TRUE,FALSE);        //send an on-bus message
                                            //to wake up mcp250x0's
   delay_ms(1000);                          //wait for node c to power-up
   while(TRUE)
   {
      write_7_segment(i);
      delay_ms(1000);
      if( ++i==10)
         i=0;

      write_c_led(GREEN, i>1);
      write_c_led(YELLOW, i>4);
      write_c_led(RED, i>7);
      delay_ms(10);
   }
}
```

- [ ] Click Compile and load the program into Node A.
- [ ] The program should display 0-9 on the LED and light the green, yellow and red LEDs on Node C, if, according to the value, is >1, >4, >7 respectively.

9 USING THE MCP250XX FOR INPUT

- The MCP250xx part used on Node C has been programmed to send a frame whenever one of the pushbuttons change value (GP4-GP6).
- The following program will read CAN bus messages looking for that specifi Node C ID. It will then light a LED depending on the button pressed.
- Add this line to ccsana.c:

  #define NODE_C_PUSHBUTTON_ID 0x303

- Click on **File> New> Source File** and enter the filename **EX9.c.**
- Type in the following program.

```
#include <ccscana.c>

void main() {
   int8 buffer[8],rx_len;
   struct rx_stat rxstat;
   int32 rx_id;

   can_init();      // always initialize the can

   can_enable_b_transfer(TRB0);  //enable buffer 0 as transmit buffer
   can_enable_b_transfer(TRB1);  //enable buffer 1 as transmit buffer
   can_enable_b_transfer(TRB2);  //enable buffer 2 as transmit buffer

   can_putd(0x100,0,0,1,TRUE,FALSE);    //send an on-bus message
                                        //to wake up mcp250x0's
   delay_ms(1000);                      //wait for node c to power-up

   while(TRUE) {
      if ( can_kbhit() ) {
         if(can_getd(rx_id, &buffer[0], rx_len, rxstat))
            if (rx_id == NODE_C_PUSHBUTTON_ID) {
               write_c_led(YELLOW, bit_test(buffer[0],4));
               write_c_led(GREEN, bit_test(buffer[0],5));
               write_c_led(RED, bit_test(buffer[0],6));
            }
      }
   }
}
```

- Click Compile and run the program.

- The write_c_led function calls send a frame to Node C to light a LED. We will now add a program to Node B to look for this same data and perform the same action at Node B.
- Click on **File> New> Source File>** and enter the filename **EX9A.C.**
- Type in the following program:

```
#include <30F4012.h>
#fuses PR,FRC_PLL16,NOWDT
#use delay(clock=117.92M)
#use rs232(baud=9600,UART1A)

#include "can-mcp2510.c"

#define RED_LED PIN_E1
#define YELLOW_LED PIN_E2
#define GREEN_LED PIN_E4
#define WRITE_REGISTER_C_ID 0x300

void main ( )
{
   int32 rx_id;
   int8 rx_len,buffer[8];
   struct rx_stat rxstat;
   int1 a,b;

   a = b = FALSE;

   can_init ( );

   while ( TRUE )
   {
      if ( can_getd ( rx_id , &buffer [ 0 ] , rx_len , rxstat ) )
      {
         if ( rx_id == WRITE_REGISTER_C_ID && buffer [ 0 ] == 0x1e )
         {
            if ( rx_id == WRITE_REGISTER_C_ID && buffer [ 0 ] == 0x1e )
            {
               if ( buffer [ 1 ] & 4 )
                  a = buffer [ 2 ];
               if ( buffer [ 1 ] & 8 )
                  b = buffer [ 2 ];

               output_bit(RED_LED,!(a==b));
               output_bit(YELLOW_LED,!(b==TRUE && a==FALSE));
               output_bit(GREEN_LED,!(a==TRUE && b==FALSE));
            }
         }
      }
   }
}
```

- Compile and load into Node B.

10 USING THE MCP250XX FOR ANALOG INPUT AND SCHEDULING DATA

- The MCP25050 can be configured for up to four analog inputs. The A/D converter is 10 bits (0-1023). The following program makes a request for the ID with A/D results 10 times per second, then waits for the frame to be sent with that ID. The following example clearly shows these features, however, it is not recommended for a real application. This program will hang if the MCP25050 does not answer.
- Click on **File> New> Source File>** and enter the filename **EX10.C.**
- Type in the following program:

```
#include <ccscana.c>

void main() {
   int1 waiting;
   int8 buffer[8],rx_len;
   struct rx_stat rxstat;
   int32 rx_id;
   int16 ad_val;

   can_init();     // always initialize the can

   can_enable_b_transfer(TRB0);

   can_putd(0x100,0,0,1,TRUE,FALSE);    //send an on-bus message
                                               //to wake up
mcp250x0's
   delay_ms(1000);                      //wait for node c to power-up

   while(TRUE) {
      delay_ms(100);
      can_putd(WRITE_REGISTER_C_ID, 0, 8, 1, TRUE, TRUE);
      waiting=TRUE;
      while(waiting) {
         if ( can_kbhit() )
            if(can_getd(rx_id, &buffer[0], rx_len, rxstat))
               if (rx_id == WRITE_REGISTER_C_ID) {
                  write_7_segment((unsigned int8)buffer[2]/26);
                  waiting=false;
               }
      }
   }
}
```

- Compile and load into Node A. Test the program by turning the Node C pot. Node A should use the A/D reading to display a number 0-9 on the Node D LED.

- The rate the data is updated to the display is determined by the delay_ms line. Try a delay_ms(1000) to get a feel for how that lag works. Then try a delay_ms(1).
- The settings on the MCP25050 have been governed by what was pre-programmed into the EEPROM. In this next program, the preprogrammed settings will be changed. This chip has the capability to send certain messages when specific, one-time events happen or when events happen, on a regular basis. The chip will be programmed to send out the analog frame roughly 10 times per second.
- Add #define NODE_C_SCHEDULED 0x301 to CCSCANA.C.
- Click on **File> New> Source File>** and enter the filename **EX10A.C.**
- Type in the following program:

```
#include <ccscana.c>

void main() {
   int8 buffer[8],rx_len;
   struct rx_stat rxstat;
   int32 rx_id;

   can_init();                        // always initialize the can

   can_enable_b_transfer(TRB0);

   can_putd(0x100,0,0,1,TRUE,FALSE);  //send an on-bus message
                                      //to wake up mcp250x0's
   delay_ms(1000);                    //wait for node c to power-up

   buffer[0]=0x2C;
   buffer[1]=0xFF;
   buffer[2]=0xD7;                    // Sched ON, For READ ADC,
clock *4096 *16 * 7
   can_putd(WRITE_REGISTER_C_ID, buffer, 3, 1, TRUE, FALSE);

   while(TRUE) {
      if ( can_kbhit() ) {
         if(can_getd(rx_id, &buffer[0], rx_len, rxstat)) {
            if (rx_id == NODE_C_SCHEDULED) {
               write_7_segment((unsigned int8)buffer[2]/26);
            }
         }
      }
   }
}
```

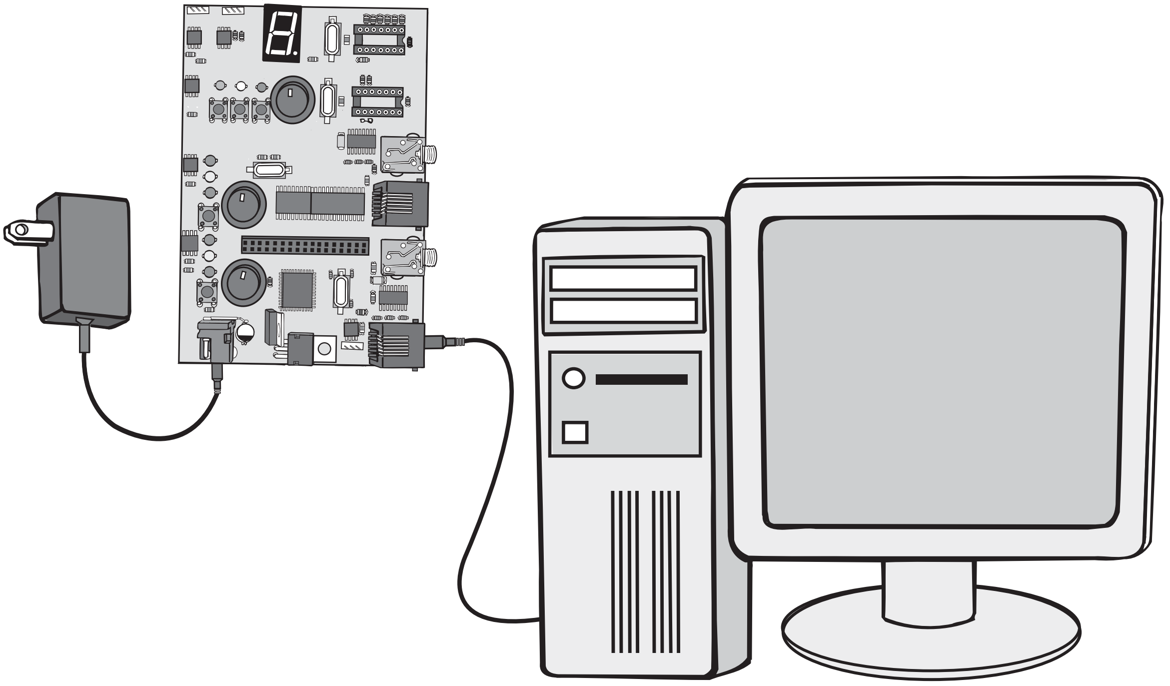
- Compile and load into Node A.

11 A CAN BUS MONITOR

- The following program is intended for Node B. It will take all frames from the CAN bus and send them over a RS-232 link. A PC must be connected to the RS-232 port to view the data. Use the Serial Port Monitor program to view the data on the RS-232 port.
- Click **File> New> Source File>** and enter the filename **EX11.C.**
- Type in the following:

```
#include <30F4012.h>
#fuses PR,FRC_PLL16,NOWDT
#use delay(clock=117.92M)
#use rs232(baud=9600,UART1A)

#include <can-mcp2510.c>

void main() {
   int32 rx_id;
   int8 i, rx_len, buffer[8];
   struct rx_stat rxstat;

   can_init();         // always initialize the can

   while(TRUE) {
      if ( can_kbhit() ) {
         if(can_getd(rx_id, &buffer[0], rx_len, rxstat)) {
            printf("%LX: (%U) ",rx_id,rx_len);
            if (!rxstat.rtr) {
               for(i=0;i<rx_len;i++)
                  printf("%X ",buffer[i]);
            }
            if (rxstat.rtr) {printf(" R ");}
            if (rxstat.err_ovfl ) {printf(" O ");}
            if (rxstat.inv) {printf(" I ");}
            printf("\r\n");
         }
      }
   }
}
```

- Compile and load this program into Node B. Load the **EX8A.C** program into Node A.
- Notice the CAN bus activity between Nodes A and C are mentioned and reported over the RS-232 port.



Sample Output:

00000300: (8) R
00000401: (0)
00000301: (0)
00000303: (2) 40 3E
00000300: (3) 1E 04 FF
00000300: (3) 1E 08 FF
00000303: (2) 40 3C
00000300: (3) 1E 04 FF
00000300: (3) 1E 08 FF

12 ADVANCED DEBUGGING

- ❑ RS-232 printf statements can be a good tool to help debug a program. It does, however, require an extra hardware setup to use. If the ICD is being used as a debugger the compiler can direct putc() and getc() through the debugger interface to the debugger screen. Change the RS232 line from Chapter 11 to the following:
```
#use rs232 (DEBUGGER, xmit = PIN_B3, rcv = PIN_B3)
```
- ❑ and add `#device ICD=TRUE`.
- ❑ **Compile** and load the program into Node B.
- ❑ Verify that userstream enabled is set to True on Debug Configure tab.
- ❑ Click GO, then click the **Monitor** tab.
- ❑ Data should appear. Confirming that the program is working.
- ❑ Stop and reset the program.
- ❑ Set a breakpoint on the line:
```
if(!rxstat.rtr){
```
- ❑ Click the debugger **Break Log** tab, check the LOG box, set the breakpoint as 1 and expression as **rxstat.rtr**. Result is the value of the number being converted.
- ❑ Click GO, then click the **Log** tab and notice that each time the breakpoint was hit the value of the **rxstat.rtr** variable was logged. In this case the breakpoint did not cause a full stop of the program, it just logged the value of the requested expression and continued on.
- ❑ Stop the program.
- ❑ Delete the breakpoint by selecting the breakpoint and click on the  icon.
- ❑ Uncheck the LOG box under the log tab.
- ❑ Set a breakpoint on the last printf() in the program.
- ❑ Enter watches for **rxstat.rtr, rxstat.err_ovfl,** and **rxstat.inv.**.
- ❑ Click GO.
- ❑ When the break is reached click on the snapshot icon: 
- ❑ Check **Time** and **Watches**, uncheck everything else.
- ❑ If a printer is connected to the PC select **Printer**, otherwise select **Unique file**.
- ❑ Click on the **Now** button.
- ❑ Notice the requested data (time and watches) are either printed or written to a file as requested.
- ❑ Click on the snapshot icon again and this time select **Append to file**, put in a filename of **EX12.TXT** and check **After each single step**.

- ❑ Check **Last C line executed** in addition to the **Time** and **Watch** selected already and close the snapshot window.
- ❑ Reset and then Step Over until the final printf() is executed.
- ❑ Use **File>Open>Any File** to find the file **EX12.TXT** (by default in the Debugger Profiles directory) after setting the file type to all files.
- ❑ Notice the log of what happened with each step over command.
- ❑ Uncheck the **After each single step** in the snapshot window.
- ❑ Click Reset then Go.
- ❑ When the break is reached click on the **Peripherals** tab and select Timer 0.
- ❑ Shown will be the registers associated with timer 0. Although this program does not use timer 0 the timer is always running so there is a value in the **TMR0** register. Write this value down.
- ❑ Clear the breakpoints and set a new breakpoint.
- ❑ Click GO.
- ❑ Check the **TMR0** register again. If the new value is higher than the previous value then subtract the previous value from the current value. Otherwise, add 256 to the current value and then subtract the previous value (because the timer flipped over).
- ❑ The number we now have is the number of clock ticks it took to execute the switch and addition. A clock tick by default is 0.2ms. Multiply your number of ticks by 0.2 to find the time in ms. Note that the timers (and all peripherals) are frozen as soon as the program stops running.

FURTHER STUDY

A *The debugger **Eval** tab can be used to evaluate a C expression. This includes assignments. Set a break before the switch statement and use the Eval window to change the operator being used. For example, type a + but change it to a - before the switch.*

B *Set a break on the switch statement and when reached, change to the C/ASM view and single step through the switch statement. Look up the instructions executed in the PIC16F876A data sheet to see how the switch statement is implemented. This implementation is dependent on the case items being close to each other. Change * to ~ and then see how the implementation changes.*

13 DATA FILTERING

- [ ] The previous program recognizes that the processor must spend time reading every frame on the CAN bus. This processing time is spent even though that node only has interest in one message type. With a large number of nodes on the CAN bus, this can cause considerable wasted processing time. The solution is to get the CAN bus controller hardware to filter the data and only bother the microcontroller with data that is of interest. The following are several popular methods for filtering.

BCAN – Basic CAN

- The system is designed such that various bits in ID are used to group common frames together. A mask and reference ID are programmed into the CAN bus controller. If (FRAME_ID & MASK) == REF_ID, the frame is saved for the microcontroller; otherwise it is discarded. It is common in a BCAN controller to assign a priority to outgoing frames. This way as the controller waits for bus time messages can be sorted.
- Advanced variations of BCAN can allow multiple masks and reference IDs to be specified.
- BCAN is the scheme used on the Microchip CAN controllers. Microchip has two buffers. One allows a mask and two reference IDs. The other allows a mask and four reference IDs.

FCAN – Full CAN

- A list of all possible IDs of interest to the microcontroller is programmed into the CAN controller. A buffer is allocated in the controller for each ID. The microcontroller can then poll for data by checking buffers of interest or program certain IDs to generate an interrupt. The same buffer scheme is used for outgoing frames. The FCAN controller can handle requests for a particular ID without microcontroller intervention.
- Consider the previous program. If we had a FCAN controller then instead of waiting for a message and then acting on it the software could just request the last frame for a given ID and use the data. The same data might be used over and over until it is replaced.
- Advanced variations of FCAN allow BCAN like masks to be applied to buffers.

- **DCAN – Direct CAN**
  This is a hybrid approach with BCAN-like masks and reference IDs, FCAN-like individual receive buffers, and a BCAN-like transmit buffer.
- **TTCAN – Time Triggered CAN**
  The bus bandwidth is split into time slots. Specific frame IDs are assigned to certain timeslots. This limits the frequency for the data and helps nodes to know when to be looking for data.

- ❑ The following program will set up filtering on the Node B data monitoring program. We will set the mask and reference ID to only monitor data to Node D.
- ❑ Load **EX8A.C** into Node A.
- ❑ Open **EX11.C** add the following after the can_init( ) line and save as **EX13.C**

```
can_set_mode(CAN_OP_CONFIG);   //must be in config mode
                               //before params can be set
can_set_id(RX0MASK,0xFF00,TRUE);
can_set_id(RX0FILTER0,0x400,TRUE);
can_set_id(RX0FILTER1,0x400,TRUE);
can_set_id(RX1MASK,0xFF00,TRUE);
can_set_id(RX1FILTER2,0x400,TRUE);
can_set_id(RX1FILTER3,0x400,TRUE);
can_set_id(RX1FILTER4,0x400,TRUE);
can_set_id(RX1FILTER5,0x400,TRUE);

can_set_mode(CAN_OP_NORMAL);
```

- ❑ Compile **EX13.C** and load into Node B. Notice that only CAN messages sent to Node D are reported over the RS232 port.

14 PIC24 AND DSPIC33 ECAN MODULES

- The newer version of Microchip's CAN module is known as the Enhanced Controller Area Network or simply ECAN. This newer module can be found on the PIC24HJxxxGP5xx, PIC24HJ256GP610, dsPIC33FJxxxGP7xx, dsPIC33FJxxxGP8xx, dsPIC33FJxxxMC5xx, dsPIC33FJxxxMC7xx, and dsPIC33FJxxxMC8xx families of microcontrollers. The CAN Bus 24 board uses the PIC24HJ256GP610. The ECAN module is completely backwards compatible with the original CAN module, however, it offers many new features which include:
    - Up to 32 buffers capable of handling eight bytes of data. Of which the first 8 buffers can be used for either transmitting or receiving.
    - 16 programmable filters.
    - Three programmable masks.
    - Buffers can be set to receive in FIFO mode.
    - Automatic RTR response function for all eight transmit buffers.
- The PIC24 and dsPIC33 ECAN Modules are capable of using up to 32 buffers each capable of handling eight bytes of data. The size of the buffer that the module uses can be set in three ways.
    1. By default the PIC24 and dsPIC33 CAN driver, can-PIC24.c, is designed to use 32 buffers.
    2. The number of buffers can be changed by adding one of the following lines before the #include <can-PIC24.C> line in your main program.
       ```
       #define CAN_BUFFER_SIZE 32
       #define CAN_BUFFER_SIZE 24
       #define CAN_BUFFER_SIZE 16
       #define CAN_BUFFER_SIZE 12
       #define CAN_BUFFER_SIZE 8
       #define CAN_BUFFER_SIZE 6
       #define CAN_BUFFER_SIZE 4
       ```
    3. The number of buffers can be changed by adding the function can_set_buffer_size(x) after the can_init() function in your main program with x being 32, 24, 16, 12, 8, 6, or 4.
- By design the PIC24 and dsPIC33 ECAN modules can only use 32, 24, 16, 12, 8, 6, or 4 buffers. If another number is used in methods 2 and 3, the ECAN driver will default the number of buffers to 32.

15 PROGRAMMABLE BUFFERS

- Both the PIC24 and dsPIC33 ECAN modules have up to eight programmable buffers. These buffers, as their names imply, can be used to either transmit or receive data across the CAN bus. In order to set the functionality of a buffer, the can_ enable_b_transfer, and the can_enable_b_receiver, functions can be used. The values of the parameters are actually binary flags making it simpler to just use the defined labels TRB0 – TRB7 where B0 is the zeroth buffer. Below are some examples of how these functions might be used.

```
can_enable_b_transfer(TRB0); // enables TRB0 as transmitter

can_enable_b_receiver(TRB5); // enables TRB5 as receiver
```

  - On reset, all of the programmable buffers are set to receive data. Therefore, the last example above would only be needed if the buffer was previously set as a transmit buffer.

16 TRANSMITTING AND RECEIVING DATA WITH ECAN

- Transmitting and receiving with ECAN is almost completely the same as with CAN. The basic can_getd and can_putd functions can still be used to transfer data and logical functions such as can_kbhit() can still be used to test if data has been received. As noted in the last chapter, in order to use the programmable buffers as transmit buffers, they must be set using the appropriate functions as all of the programmable buffers default to receive on reset.
- Click **File> New> Source File>** and enter filename **EX16.C.**
- Type in the following program:

```
#include <30F4012.h>
#fuses FRC,FRC_PLL16,NOWDT
#use delay(clock=117.92M)
#use rs232(baud=9600,UART1A)

#include <can-mcp2510.c>

void main()
{
   int32 rx_id;
   int8 rx_len;
   struct rx_stat stat;
   int8 data[8]={7,6,5,4,3,2,1,0};
   int8 receive[8];

   can_init();            // always initialize the can

   while(TRUE)
   {
      if(can_kbhit())
      {
         can_getd(rx_id,receive,rx_len,stat);
         printf("Data has been received\n\r");
         can_putd(0x600,data,8,3,TRUE,FALSE);
         printf("Data has been sent\n\r");
      }
      else
      {
         printf("no data found\n\r");
      }
      delay_ms(3000);
   }
}
```

- Click Compile and load into Node B.

- This program is a simple echo program, it enters an infinite loop and then tests to see if there is data waiting in any of the buffers. If there is, it then loads that data, prints a statement acknowledging that it has loaded the data and then puts some different data onto the bus. After acknowledging that the data has been sent, the if statement exits, and there is a three second delay before the cycle starts again.
- Add this line to ccsana.c:

  #use rs232(baud=9600, UART1)
- The following program will send the first program some data and listen for a response.
- Click **File> New> Source File>** and enter filename **EX16A.C**
- Type in the following program:

```
#include "CCSCANA.c"

void main ( )
{
   int32 rx_id;
   int8 rx_len,i;
   struct rx_stat stat;
   int8 data[8]={15,14,13,12,11,10,9,8};
   int8 receive[8];

   can_init();            // always initialize the can

   can_enable_b_transfer(TRB0);

   while(TRUE)
   {
      for(i=0;i<8;i++)
      {
         printf("%i ",data[i]);
      }
      printf("\n\rIs being placed on the bus with id 0x00000500\n\r\n\r");

      can_putd(0x500,data,8,3,TRUE,FALSE);

      while(!can_kbhit());               // wait for a response

      can_getd(rx_id,receive,rx_len,stat);

      for(i=0;i<8;i++)
      {
         printf("%i ",receive[i]);
      }
      printf("\n\rWas received with id 0x%Lx\n\r\n\r",rx_id);

      delay_ms(3000);
   }
}
```

- Click on compile and load into Node A.
- The following is sample output from node A.

      15 14 13 12 11 10 9 8
      Is being placed on the bus with id 0x00000500

      7 6 5 4 3 2 1 0
      Was received with id 0x00000600

- There are certain instances in which the program may only need to access the transmit buffer once every few seconds; and, therefore will only ever use one transmit buffer. There are functions included in the ECAN device library that will allow the user to set up a specific transmit register.
- Replace the following line of code:

      can_putd(0x500,data,8,3,TRUE,FALSE);
      with this.
      can_trb0_putd(0x500,data,8,3,TRUE,FALSE);

- This will attempt to place the data into the zeroth transmit register. If the buffer happens to be transmitting or full, the function will return false otherwise it will return true. Each buffer has a function associated with it. These functions are as follows.

      can_trb0_putd
      can_trb1_putd
      can_trb2_putd
      can_trb3_putd
      can_trb4_putd
      can_trb5_putd
      can_trb6_putd
      can_trb7_putd

- These eight functions will write to the programmable buffers. It should be noted again, that in order to use these functions, each associated buffer must be set to transmit mode.
- The purpose of these functions is mainly to reduce the amount of program memory dedicated to placing data on the bus. In the case of the original CAN bus, there were only three buffers to check, however, now there are eight. If only one transmit register is needed, it is much more efficient not to test each buffer and simply write to the buffer that is to be used.

- Recall that the PIC24 and dsPIC33 ECAN modules have 16 filters and 3 masks. Each filter can be associated to mask 0, mask 1, mask 2 or no mask, and each filter can be associated to buffers 0-14 or the FIFO buffer. The mask register is used to determine which bits of the incoming ID the filter should be applied two. Therefore, if the mask had a value of 0x01, only the least significant bit would have the filter applied to it. The filter is used as a reference to determine which IDs to accept and which to reject. If, for instance, the filter was 0xFF, only numbers with the value of 0xFF would be accepted, unless the mask only applied that filter to certain bits. In this case, the bits that the filter was applied to would need to be high in order to be accepted by the filter.
- Also note that many filters can be associated with a single buffer, but multiple buffers can not be associated with the same filter.
- The steps then for setting up filters on the PIC24 and dsPIC33 are as follows.
  1.Load masks and filters with desired Ids using the can_set_id function.
  2.Associate each used filter with a mask using the can_associate_filter_to_mask function.
  3.Associate each used filter with a buffer using the can_associate_filter_to_buffer function.
- Click **File> New> Source File>** and enter filename **EX17.C**.
- Type in the following program:

```
#include <30F4012.h>
#fuses FRC,FRC_PLL16,NOWDT
#use delay(clock=117.92M)
#use rs232(baud=9600,UART1A)

#include <can-mcp2510.c>

void main()
{
   int8 data[8]={7,6,5,4,3,2,1,0};

   can_init();            // always initialize the can

   while(TRUE){
      can_putd(0x600,data,8,3,TRUE,FALSE);
      delay_ms(1000);
      can_putd(0x700,data,8,3,TRUE,FALSE);
      delay_ms(1000);
      can_putd(0x800,data,8,3,TRUE,FALSE);
      delay_ms(1000);
      can_putd(0x900,data,8,3,TRUE,FALSE);
      delay_ms(1000);
   }
}
```

- Click Compile and load into Node B.
- This is a simple transmitter program that sends out data with several IDs.

17 USING FILTERS (CONT.)

- ❑ Next, copy, compile and load the following receiver program into node A.
- ❑ Click **File> New> Source File>** and enter filename **EX17A.C**
- ❑ Type in the following program:

```
#include "CCSCANA.c"

void main ( )
{
   int32 rx_id;
   int8 rx_len,i;
   struct rx_stat stat;
   int8 receive[8];

   can_init();              // always initialize the can

   while(TRUE)
   {
      if(can_kbhit()) // wait for a response
      {
         can_getd(rx_id,receive,rx_len,stat);
         for(i=0;i<8;i++)
         {
            printf("%i ",receive[i]);
         }
         printf("\n\rwas received with id %Lx\n\r\n\r",rx_id);
      }
   }
}
```

- ❑ Click Compile and load into Node A.
- ❑ This program simply listens to the bus and prints out the data and the ID as they are received. The following is sample output from this code.

7 6 5 4 3 2 1 0
was received with id 00000600

7 6 5 4 3 2 1 0
was received with id 00000700

7 6 5 4 3 2 1 0
was received with id 00000800

7 6 5 4 3 2 1 0
was received with id 00000900

- Try to filter out all of the IDs except for 0x600. To do this, add the following code right after can_init().

```
can_set_id(&C1RXM0,0xFF00,TRUE);
can_set_id(&C1RXF0,0x600,TRUE);
can_associate_filter_to_mask(F0BP,ACCEPTANCE_MASK_0);
can_associate_filter_to_buffer(ATRB0,F0BP)
```

- The first line sets up the mask ID. In this case, only bytes two and three will have the filter applied to them. This works well because the only addresses that are dealt with are 0x600, 0x700, 0x800, and 0x900. If expecting an address such as 0x5432, it would probably be best to load the mask with the value 0xFFFF.
- The second line sets the filter IDs of filter 0 to 0x600.
- The third line associates filter 0 with acceptance mask 0. There are four possible masks that can be associated with a filter. These are as follows.

```
ACCEPTANCE_MASK_0
ACCEPTANCE_MASK_1
ACCEPTANCE_MASK_3
NO_MASK
```

- The fourth line associates filter 0 with the programmable buffer 0.
- After compiling and running the code, the output should look something like this.

```
7 6 5 4 3 2 1 0
was received with ID 00000600

7 6 5 4 3 2 1 0
was received with ID 00000600

7 6 5 4 3 2 1 0
was received with ID 00000600
```

- As can be seen by the output, only 0x600 IDs are allowed into the receive buffers.

NOTES

- The function can_init() disables all of the filters except filter 0 and associates it with the FIFO buffer. So if another filter is desired to be used instead of filter 0, filter 0 will need to be disabled with the can_disable_filter() function and the other filter will need to be enabled with the can_enable_filter() function. For example if filter 10 was desired with buffer 8 the code would be as follows.

```
can_disable_filter(FLTEN0);
can_set_id(&C1RXM0,0xFF00,TRUE);
can_set_id(&C1RXF10,0x600,TRUE);
can_associate_filter_to_mask(F10BP,ACCEPTANCE_MASK_0);
can_associate_filter_to_buffer(ARB8,F10BP);
can_enable_filter(FLTEN10);
```

18 RECEIVING IN FIFO MODE

- The ECAN module provides a first in first out (FIFO) functional mode that allows received data to be retrieved without having to manually look at each buffer to see if it is full. When data comes into a register, an internal pointer available to read though one of the ECAN registers, points to the buffer that has the data. If more data were to come in while the data was being processed, the pointer would point to the first buffer that had been filled. For example, if buffers zero, five, four, and then seven were filled in that order, the pointer would first point to buffer zero. Once buffer zero had been read, the pointer would point to buffer five. Once buffer five had been read the pointer would point to four and so on until there were no full registers. All of the described functionality is taken care of in the device drivers, however it is beneficial to understand how the process works.
- The FIFO buffer can consist of up to 32 buffers each capable of containing 8 bytes of data. The first eight buffers can be programmed as either transmit or receive buffers. Depending on the number of buffers being used in the ECAN module, the FIFO buffer can be anywhere between zero and thirty two receive buffers long. The length of the FIFO buffer is determined by the number of buffers the ECAN module is using and which of the programmable buffers is set to be a transmit buffer. The highest programmable buffer configured to transmit is the cut off point for the FIFO buffer. For example, if the highest programmable transmit buffer was TRB3 and the number of buffers being use is thirty two, than the FIFO buffer would consist of the four programmable buffers TRB4, TRB5, TRB6, TRB7 along with RB8-RB31, creating a 28 buffer long FIFO buffer. If TRB7 was a transmit buffer, than the FIFO buffer would be 24 buffers long, and if all of the programmable buffers were set to receive, then the FIFO buffer would be 32 buffers deep, the maximum size.
- Click **File> New> Source File>** and enter filename **EX18.C**.

```
#include <30F4012.h>
#fuses FRC,FRC_PLL16,NOWDT
#use delay(clock=117.92M)
#use rs232(baud=9600,UART1A)

#include "can-mcp2510.c"

void main()
{
   int8 data[8]={7,6,5,4,3,2,1,0};

   can_init();
   while(TRUE){
      can_putd(0x100,data,8,3,TRUE,FALSE);
      can_putd(0x200,data,8,3,TRUE,FALSE);
      can_putd(0x300,data,8,3,TRUE,FALSE);
      can_putd(0x400,data,8,3,TRUE,FALSE);
      delay_ms(3000);
   }
}
```

- Click Compile and load into Node B.

- This program simply sends four consecutive data frames, and then delays three seconds before repeating. We will use these data frames to demonstrate how the FIFO system works and how the length of the FIFO can be changed.
- Click **File> New> Source File>** and enter filename **EX18A.C**.
- Type in the following program:

```
#include "CCSCANA.c"

void main()
{
   int32 rx_id;
   int8 rx_len,i;
   struct rx_stat stat;
   int8 receive[8];

   can_init();              // always initialize the can

   can_set_buffer_size(8);

   while(TRUE)
   {
      if(can_kbhit()) // wait for a response
      {
         can_fifo_getd(rx_id,receive,rx_len,stat);
         for(i=0;i<8;i++)
         {
            printf("%i ",receive[i]);
         }
         printf("\n\rid = %Lx\n\r",rx_id);
         printf("buffer = %i\n\r\n\r",stat.buffer);
      }
   }
}
```

- Click Compile and load into Node A.

- This code uses the can_fifo_getd function as apposed to the can_getd function. This new function uses the pointer described above to retrieve the data from the buffer in stead of polling each buffer to see if data has been received. This significantly reduces the amount of program memory used and cuts the amount of time that it takes to execute the function.

18 RECEIVING IN FIFO MODE (CONT.)

- ❑ Below is a sample of the first six seconds of output.

```
7 6 5 4 3 2 1 0
id = 00000100
buffer = 0
□
7 6 5 4 3 2 1 0
id = 00000200
buffer = 1

7 6 5 4 3 2 1 0
id = 00000300
buffer = 2
□
7 6 5 4 3 2 1 0
id = 00000400
buffer = 3
□
7 6 5 4 3 2 1 0
id = 00000100
buffer = 4
□
7 6 5 4 3 2 1 0
id = 00000200
buffer = 5
□
7 6 5 4 3 2 1 0
id = 00000300
buffer = 6
□
7 6 5 4 3 2 1 0
id = 00000400
buffer = 7
```

- ❑ Notice the output that eight data frames where received and that the FIFO system filled the receive buffers in order from zero to seven.
- ❑ Add the following line to the program just after the can_set_buffer_size function call.

```
can_enable_b_transfer(TRB3);
```

- ❑ This not only enables the TRB3 programmable buffer to be a transmit buffer, it also cuts the FIFO buffer by half because now the start of the FIFO buffer is set to buffer TRB4.

- The output for the modified program is as follows.

```
7 6 5 4 3 2 1 0
id = 00000100
buffer = 4

7 6 5 4 3 2 1 0
id = 00000200
buffer = 5

7 6 5 4 3 2 1 0
id = 00000300
buffer = 6

7 6 5 4 3 2 1 0
id = 00000400
buffer = 7
```

- In order to get the FIFO buffer to its maximum size, any programmable buffers that need to be configured as transmit buffers should use the lower buffers. For example if two transmit buffers are needed, they should be set to programmable buffers TRB0 and TRB1 because that way TRB2 through TRB7 can be used as receive buffers for the FIFO buffer.
- Also, the previous example set the buffer size to 8 with the can_set_buffer_size function. The FIFO buffer can also be increased in size by setting the buffer size of the ECAN module to 12, 16, 24, or 32. The method for doing this is described in section 14 of the manual.

19 USING FILTERS IN FIFO MODE

- Filters in FIFO mode work a little differently than filters not in FIFO. Previously, the filters were associated with a given buffer dynamically. The buffers where thought of as individual and therefore could be associated with individual filters. In FIFO mode, however, buffers are not thought of as individual buffers, but rather as part of the entire FIFO buffer. It would not make sense to associate a filter to a buffer because that buffer may never be needed by the FIFO system. Therefore, any filter that is enabled, needs to be associated with all of the buffers in the FIFO buffer. Each individual filter, however, can still be dynamically associated with one of the three masks.
- The process to set up the filters in FIFO mode is as follows.
  1. Enable any filters that will be needed and disable all that will not using the can_enable_filter and can_disable_filter functions.
  2. Set the mask and filter IDs to the needed values using the can_set_id function.
  3. Associate each filter to the required masks using the can_associate_filter_to_mask function.
  4. Associate each filter to the FIFO buffer using the can_associate_filter_to_buffer function.
- Once this has been done, only IDs that match any of the filter values will be allowed into the FIFO buffer.
- Add the following code to EX18 after the can_enable_b_transfer function.

```
can_enable_filter(FLTEN0);
can_enable_filter(FLTEN1);

can_set_id(&C1RXM0,0xff00,TRUE);
can_set_id(&C1RXF0,0x400,TRUE);
can_set_id(&C1RXF1,0x800,TRUE);

can_associate_filter_to_mask(ACCEPTANCE_MASK_0,F0BP);
can_associate_filter_to_mask(ACCEPTANCE_MASK_0,F1BP);

can_associate_filter_to_buffer(AFIFO,F0BP);  // associates filter 0 to FIFO buffer
can_associate_filter_to_buffer(AFIFO,F1BP);  // associates filter 1 to FIFO buffer
```

- Make sure that the node B program from chapter 18 is still running on node B. Compile and load the node A program into node A. After opening the Serial Port Monitor interface program, the output should look like the following.

7 6 5 4 3 2 1 0 id = 00000400 buffer = 4	7 6 5 4 3 2 1 0 id = 00000800 buffer = 5
7 6 5 4 3 □ id = 00000400 buffer = 6	id = 00000800 buffer = 7

20 USING AUTO-RTR

- Until now, all received data needed to be read, processed, and responded to entirely in software. An optimal situation would be if it was possible to load data into one of the transmit buffers, give that buffer an ID, and then tell that buffer which ID to respond too when a remote transmission was requested. Once all of these buffer parameters have been set, the hardware would then do the work of filtering the ID and responding to the received message. This is exactly what the auto-RTR functionality of the ECAN module is for and it is available on any of the programmable buffers.
- The following is a list of steps that need to be taken to set up one of the buffers to automatically respond to remote requests.

  1. Configure the desired programmable buffer to be a transmit buffer using the `can_enable_b_transfer` function.

  2. Enable any filters that are needed and disable any that are not using the `can_enable_filter and can_disable_filter` functions.

  3. Set the ID of the masks, the filters, and the transmit buffers that will be used, using the `can_set_id and can_set_buffer_id` functions.

  4. Associate the filter to a mask and the filter to the buffer using the `can_associate filter_to_mask` and `can_associate_filter_to_buffer` functions.

  5 Load the desired data into the desired transmit buffer using the `can_load_rtr` function.

  6 .Finally, enable the desired transmit buffer as an RTR buffer using the `can_ enable_rtr` function.

- To demonstrate the auto-RTR functionality, click **File> New> Source File>** and enter filename **EX20.C**.
- Click **File> New> Source File>** and enter filename **EX16A.C**
- Type in the following program:

```
#include <30F4012.h>
#fuses PR,FRC_PLL16,NOWDT
#use delay(clock=117.92M)
#use rs232(baud=9600,UART1A)

#include <can-mcp2510.c>
```

(continued...)

USING AUTO-RTR (CONT.)

(...continued)

```
void main()
{
   int32 rx_id;
   int8 rx_len,i;
   struct rx_stat stat;
   int8 data[8]={7,6,5,4,3,2,1,0};
   int8 receive[8];
   can_init();             // always initialize the can

   while(TRUE)
   {
      can_putd(0x500,data,8,3,TRUE,TRUE);
      delay_ms(1000);
      if(can_kbhit())
      {
         can_getd(rx_id,receive,rx_len,stat);
         printf("data received!\n\r");
         for(i=0;i<8;i++)
            printf("%i ",receive[i]);
         printf("\n\r%Lx\n\r",rx_id);
      }
      else
      {
         printf("data not received.\n\r");
      }

   }
}
```

- ❑ Click Compile and laod into Node B.
- ❑ This program simply puts data onto the CAN bus and then checks to see if anything was sent back. If data was sent back, it will print the data and the ID of the sender, if not, a message will be displayed. Notice that the last parameter of the can_putd function is now set to true. This tells the function that the bit frame should request a remote response instead of simply sending the data. This would be like getting a return request in an email or a letter, it simply tells the receiver that the sender would like a response to the message.
- ❑ Next, we will write a program for node A that will respond to the RTR messages being sent from node B.
- ❑ Click **File> New> Source File>** and enter filename **EX20A.C**.

- Type in the following program:

```
#include "CCSCANA.c"

void main ( )
{
   int32 rx_id;
   int8 rx_len,i;
   struct rx_stat stat;
   int8 data[8]={15,14,13,12,11,10,9,8};
   int8 receive[8];

   can_init();            // always initialize the can

   can_enable_b_transfer(TRB1);
   can_disable_filter(FLTEN0);
   can_enable_filter(FLTEN12);

   can_set_id(&C1RXM0,0xff00,TRUE);
   can_set_id(&C1RXF12,0x500,TRUE);
   can_set_buffer_id(TRB1,0x600,TRUE);

   can_associate_filter_to_mask(ACCEPTANCE_MASK_0,F12BP);
   can_associate_filter_to_buffer(ATRB1,F12BP);

   can_load_rtr(TRB1,data,8);

   can_enable_rtr(TRB1);

   while(TRUE)
   {
   }
}
```

- Click Compile and load into Node A.
- This program simply follows the six steps listed above and then enters an infinite loop which does nothing. All receive and transmit work is done completely in hardware.
- Insert the serial cable into the jack on node B and open the serial port interface. As the program is loaded into node A, the output should look something like this:

  data not received.
  data not received.
  data not received.
  data received!
  15 14 13 12 11 10 9 8
  00000600
  data received!
  15 14 13 12 11 10 9 8
  00000600
  data received!
  15 14 13 12 11 10 9 8
  00000600

21 USING ECAN INTERRUPTS

- Another useful function of the PIC24 and dsPIC33 ECAN module is the ability to interrupt the processor because of certain ECAN events. This is useful because it allows the processor to act on these events when they happen without the need to poll for them in the main loop.
- The following is a list of steps that need to be taken to set up the microcontroller to interrupt on a specified ECAN event:

  1. Select the desired interrupt event with the `can_enable_interrupts` function. The available interrupt events are:
     - i. Transmit Buffer Interrupt
     - ii. Receive Buffer Interrupt
     - iii. Receive Buffer Overflow Interrupt
     - iv. FIFO Almost Full Interrupt
     - v. Error interrupt
     - vi. Wake-Up Interrupt
     - vii. Invalid Message Received Interrupt
  2. Enable the microcontroller's global interrupt with the `enable_interrupts` function.
  3. Enable the microcontroller's overall CAN interrupts with the `enable_interrupts` function.
- To demonstrate the Interrupt functionality, click **File> New> Source File>** and enter filename **EX21.C**.
- Type in the following program:

```
#include <30F4012.h>
#fuses PR,FRC_PLL16,NOWDT
#use delay(clock=117.92M)
#use rs232(baud=9600,UART1A)

#include <can-mcp2510.c>

void main()
{
   int32 rx_id;
   int8 rx_len,i;
   struct rx_stat stat;
   int8 data[8]={7,6,5,4,3,2,1,0};
   int8 receive[8];

   can_init();             // always initialize the can

   while(TRUE)
   {
```

(continued...)

(...continued)

```
        can_putd(0x500,data,8,3,TRUE,FALSE);
        delay_ms(1000);
        if(can_kbhit())
        {
         can_getd(rx_id,receive,rx_len,stat);
           printf("Data received!\n\r");
           for(i=0;i<8;i++)
              printf("%i ",receive[i]);
           printf("\n\r%Lx\n\r",rx_id);
        }
        else
        {
           printf("No data received.\n\r");
        }
     }
  }
```

- ❑ Click Compile and load into Node B.
- ❑ This program simply puts data onto the CAN bus and then checks to see if anything was sent back. If data was sent back, it will print the data and the ID of the sender, if not, a message will be displayed.
- ❑ Next, is a program for node A that will use the receive buffer interrupt to receive the data, add ten to each of the values and then send the message back with id 0x1000.
- ❑ Click **File> New> Source File>** and enter filename **EX21.C**.
- ❑ Type in the following program:

```
#include "CCSCANA.c"

#INT_CAN1
void can_int(void)
{
   int32 rx_id;
   int8 rx_len,i;
   struct rx_stat stat;
   int8 data[8];

   can_getd(rx_id,data,rx_len,stat);
```

(continued...)

(...continued)

```
   for(i=0;i<8;i++)
      data[i]+=10;

   can_putd(0x1000,data,8,3,TRUE,FALSE);
}
void main()
{
   can_init();                    // always initialize the can

   can_enable_b_transfer(TRB1);   // enable buffer 1 as transmit buffer

   can_enable_interrupts(RB);            //selects which interrupt
event to use with the overall CAN interrupt

   enable_interrupts(INTR_GLOBAL);
   enable_interrupts(INT_CAN1);          //enables the overall CAN
interrupts

   while(TRUE)
   {
   }
}
```

- ❑ Click compile and load into Node A.
- ❑ This main program simply follows the three steps listed above and then enters an infinite loop which does nothing. All receive and transmit work is done in the interrupt service routine.
- ❑ Insert the serial cable into the jack on node B and open the serial port interface. As the program is loaded into node A, the output should look something like this:

Data received!
17 16 15 14 13 12 11 10
00001000
Data received!
17 16 15 14 13 12 11 10
00001000
Data received!
17 16 15 14 13 12 11 10
00001000
Data received!
17 16 15 14 13 12 11 10
00001000

22 CONNECTING TO AN EXTERNAL CONTROLLER

- [ ] **PHYSICAL**

As previously noted, there is no standard physical interface. The PCA82C251 chips used on the prototype board use a popular 2-wire CAN bus. Connections can be made directly from the prototyping board to an external CAN bus via the 3-pin connector at the top of the board (CANL, CANH and Ground). When using this connection over some distance, a 120 ohm resistor should be put on both ends of the bus. This driver chip can handle up to 110 nodes and a total bus length of 100 feet. The bus can be much longer if a slow-bit time is used.

An extra driver chip has been installed on the prototype board. This allows for an easy connection to an external CAN controller that has TTL output. The three pin connection has Transmit, Receive and Ground connections to the spare PCA82C251 chip.

Some CAN Transceivers

		Nodes	Speed	Fault Tolerant	
Philips	PCA82C251	110	1 meg	NO	
	PCA82C252	15	125k	YES	
	TJA1054	32	125k	YES	Low EMC
Maxim	MAX3058	32	1 meg	NO	
	MAX3050	32	2 meg	NO	
	MAX3054	32	250k	YES	
TI	SN65LBC031		500k	NO	
	SN65HVD251	120	1 meg	NO	
	SN65HVD232	120	1 meg	NO	3.3V

- **TIMING**

All nodes on the bus must have the same target bit time. The fastest time allowed by the PCA82C251 is 1 million bits per second.

A single-bit time is divided into four segments:

Sync period
Propagation period (allow for delays between nodes)
Phase 1 period
Phase 2 period

The data is sampled for the bit between phase 1 and phase 2.
Each of the four segment times may be programmed in terms of a base time (Time Quanta or Tq)..

The baud rate settings are made in the .h files (like can-PIC24.h). The following settings have been made:

Sync period = 1 Tq
Propagation period = 3 Tq
Phase 1 period = 6 Tq
Phase 2 period = 6 Tq

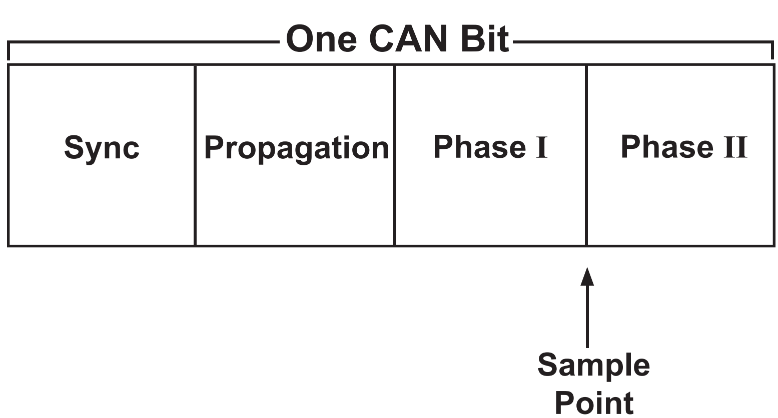
The total bit time is therfor 16 Tq.

Tq is set via the prescaller. The formula is:

Tq = (2 x (prescaller+1))/(clock/2)

Use a clock of 20 mhz and have the prescaller set to 4. Therefore:
Tq = (2 x (4+1))/20000000 = 0.1 us

The bit time is 16 us or 62.5K.



References

This booklet is not intended to be a tutorial for the C programming language. It does attempt to cover the basic use and operation of the development tools. There are some helpful tips and techniques covered, however, this is far from complete instruction on C programming. For the reader not using this as a part of a class and without prior C experience the following references should help.

Exercise	PICmicro® MCU C: An introduction to Programming the Microchip PIC® in CCS by Nigel Gardner	The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie (2nd ed.)
3	1.1 The structure of C Programs 1.2 Components of a C Program 1.3 main() 1.5 #include 1.8 constants 1.11 Macros 1.13 Hardware Compatibility 5.5 While loop 9.1 Inputs and Outputs	1.1 Getting Started 1.4 Symbolic Constants 3.1 Statements and Blockx 3.5 Loops 1.11 The C Preprocessor
4	1.7 Variables 1.10 Functions 2.1 Data Types 2.2 Variable Declaration 2.3 Variable Assignment 2.4 Enumeration 3.1 Functions 3.4 Using Function Arguments 4.2 Relational Operators 5.7 Nesting Program Control Statements 5.10 Switch Statement	1.2 Variables and Arithmetic Expr 2.1 Variable Names 2.2 Data Types and Sizes 2.3 Constants 2.4 Declarations 2.6 Relational and Logical Operators 3.4 Switch 1.7 Functions 1.8 Arguments 4.1 Basics of Functions
5	4.3 Logical Operators 4.4 Bitwise Operators 4.5 Increment and Decrement 5.1 if Statements 5.2 if-else Statements 9.3 Advanced BIT Manipulation	3.2 if-Else 2.8 Increment and Decrement Ops 2.90 Bitwise Operators
6	4.1 Arithmetic Operators	2.5 Arithmetic Operators
7	9.5 A/D Conversion	3.3 Else

8	5.4 For Loop 6.1 One-Dimensional Arrays	1.3 The For Statement 1.6 Arrays 2.10 Assignments Operators and Exp
10	1.6 printf Function 9.6 Data Comms/RS-232	1.5 Character Input and Output 2.6 Loops-Do-While 7.1 Standard Input and Output 7.2 Formatted Output - printf
11	6.2 Strings 6.4 Initializing Arrays 8.1 Introduction to Structures	7.9 Character Arrays 6.1 Basics of Structures 6.3 Arrays of Structures
13	9.4 Timers	
14	2.6 Type Conversion 9.11 Interrupts	2.7 Type Conversions
16	9.8 SPI Communications	
17	9.7 I^2C Communications	
18	5.2 ? Operator	2.11 Conditional Expressions
19	4.6 Precedence of Operators	2.12 Precedence and Order Eval

On The Web

Comprehensive list of PIC® MCU Development tools and information	www.mcuspace.com
Microchip Home Page	www.microchip.com
CCS Compiler/Tools Home Page	www.ccsinfo.com
CCS Compiler/Tools Software Update Page	www.ccsinfo.com click: Support → Downloads
C Compiler User Message Exchange	www.ccsinfo.com/forum
Device Datasheets List	www.ccsinfo.com click: Support → Device Datasheets
C Compiler Technical Support	support@ccsinfo.com

Other Development Tools

EMULATORS

The ICD used in this booklet uses two I/O pins on the chip to communicate with a small debug program in the chip. This is a basic debug tool that takes up some of the chip's resources (I/O pins and memory). An emulator replaces the chip with a special connector that connects to a unit that emulates the chip. The debugging works in a simulator manner except that the chip has all of its normal resources, the debugger runs faster and there are more debug features. For example an emulator typically will allow any number of breakpoints. Some of the emulators can break on an external event like some signal on the target board changing. Some emulators can break on an external event like some that were executed before a breakpoint was reached. Emulators cost between $500 and $3000 depending on the chips they cover and the features.

DEVICE PROGRAMMERS

The ICD can be used to program FLASH chips as was done in these exercises. A stand alone device programmer may be used to program all the chips. These programmers will use the .HEX file output from the compiler to do the programming. Many standard EEPROM programmers do know how to program the Microchip parts. There are a large number of Microchip only device programmers in the $100-$200 price range. Note that some chips can be programmed once (OTP) and some parts need to be erased under a UV light before they can be re-programmed (Windowed). CCS offers the Mach X which is a stand-alone programmer and can be used as an in-circuit debugger.

PROTOTYPING BOARDS

There are a large number of Prototyping boards available from a number of sources. Some have an ICD interface and others simply have a socket for a chip that is externally programmed. Some boards have some advanced functionality on the board to help design complex software. For example, CCS has a Prototyping board with a full 56K modem on board and a TCP/IP stack chip ready to run internet applications such as an e-mail sending program or a mini web server. Another Prototyping board from CCS has a USB interface chip, making it easy to start developing USB application programs.

SIMULATORS

A simulator is a program that runs on the PC and pretends to be a microcontroller chip. A simulator offers all the normal debug capability such as single stepping and looking at variables, however there is no interaction with real hardware. This works well if you want to test a math function but not so good if you want to test an interface to another chip. With the availability of low cost tools, such as the ICD in this kit, there is less interest in simulators. Microchip offers a free simulator that can be downloaded from their web site. Some other vendors offer simulators as a part of their development packages.

CCS Programmer Control Software

The CCSLOAD software will work for all the CCS device programmers and replaces the older ICD.EXE and MACHX.EXE software. The CCSLOAD software is stand-alone and does not require any other software on the PC. CCSLOAD supports ICD-Sxx, ICD-Uxx, Mach X, Load-n-Go, and PRIME8.

Powerful Command Line Options in Windows and Linux

- Specify operational settings at the execution level
- Set-up software to perform, tasks like save, set target Vdd
- Preset with operational or control settings for user

Easy to use Production Interface

- Simply point, click and program
- Additions to HEX file organization include associating comments or a graphic image to a file to better ensure proper file selection for programming
- Hands-Free mode auto programs each time a new target is connected to the programmer
- PC audio cues indicate success and fail

Extensive Diagnostics

- Each target pin connection can be individually tested
- Programming and debugging is tested with known good programs
- Various PC driver tests to identify specific driver installation problems

Enhanced Security Options

- Erase chips that failed programming
- Verify protected code cannot be read after programming
- File wide CRC checking

Automatic Serial Numbering Options

- Program memory or Data EEPROM
- Incremented, from a file list or by user prompt
- Binary, ASCII string or UNICODE string

CCS IDE owners can use the CCSLOAD program with:

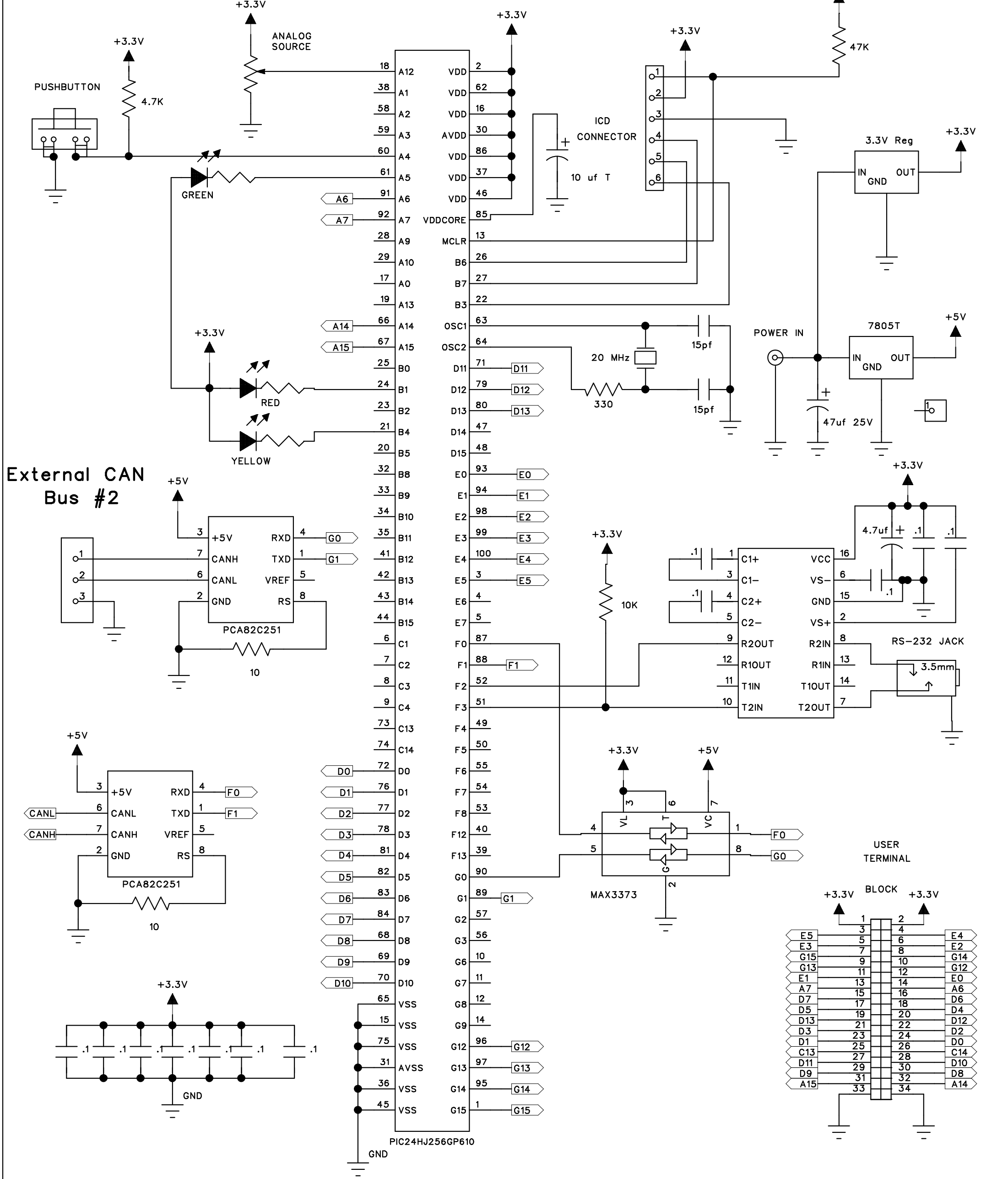
- MPLAB®ICD 2/ICD 3
- MPLAB®REAL ICE™
- **All CCS programmers and debuggers**

How to Get Started:

Step 1: *Connect Programmer to PC and target board. Software will auto-detect the programmer and device.*
Step 2: *Select Hex File for target board.*
Step 3: *Select Test Target. Status bar will show current progress of the operation.*
Step 4: *Click "Write to Chip" to program the device.*

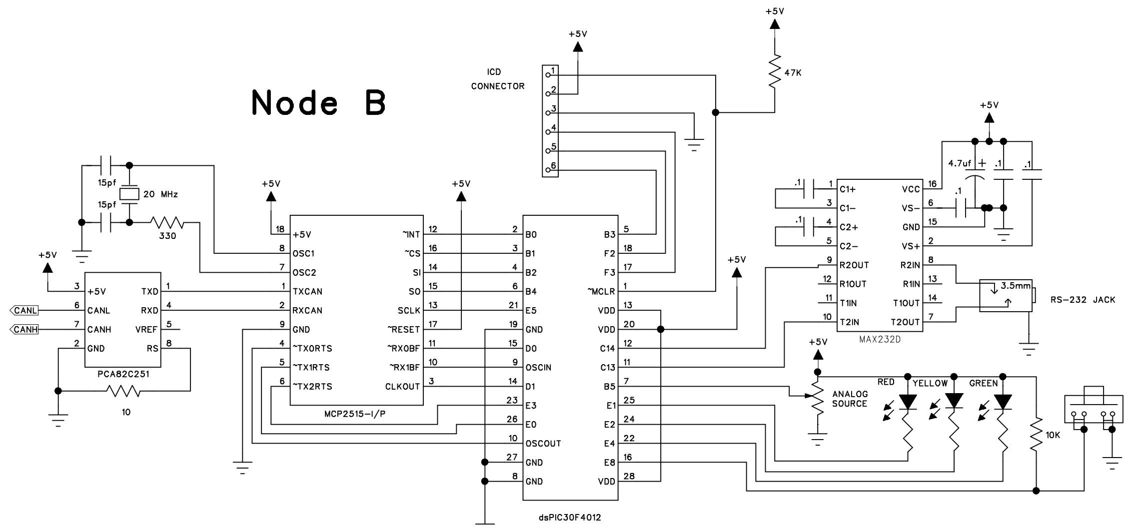
Use the Diagnostics tab for troubleshooting or the ccsload.chm help file for additional assistance.

Node A



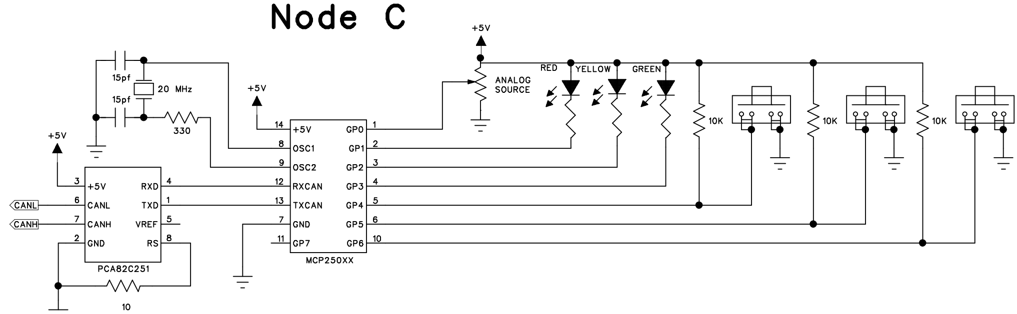
PUSHBUTTON
4.7K
ANALOG SOURCE
GREEN
RED
YELLOW
ICD CONNECTOR
10 uf T
47K
3.3V Reg
IN GND OUT
POWER IN
7805T
47uf 25V
20 MHz
15pf
330
External CAN Bus #2
PCA82C251
+5V
CANH
CANL
GND
RXD
TXD
VREF
RS
10
CANL
CANH
10K
RS-232 JACK
3.5mm
C1+
C1−
C2+
C2−
R2OUT
R1OUT
T1IN
T2IN
VCC
VS−
GND
VS+
R2IN
R1IN
T1OUT
T2OUT
4.7uf
MAX3373
USER TERMINAL BLOCK
PIC24HJ256GP610
GND
+3.3V
+5V

Node B



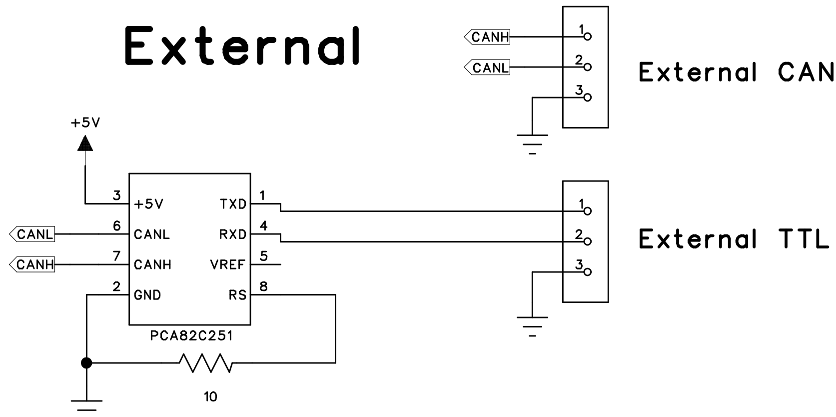
ICD CONNECTOR
+5V
47K
15pf
20 MHz
330
PCA82C251
+5V
TXD
CANL
RXD
CANH
VREF
GND
RS
10
MCP2515-I/P
OSC1
OSC2
TXCAN
RXCAN
~TX0RTS
~TX1RTS
~TX2RTS
~INT
~CS
SI
SO
SCLK
~RESET
~RX0BF
~RX1BF
CLKOUT
dsPIC30F4012
B0
B1
B2
B4
E5
D0
OSCIN
D1
E3
E0
OSCOUT
B3
F2
F3
~MCLR
VDD
C14
C13
B5
E1
E2
E4
E8
MAX232D
C1+
C1-
C2+
C2-
R2OUT
R1OUT
T1IN
T2IN
VCC
VS-
VS+
R2IN
R1IN
T1OUT
T2OUT
4.7uf
.1
3.5mm
RS-232 JACK
ANALOG SOURCE
RED
YELLOW
GREEN
10K

Node C

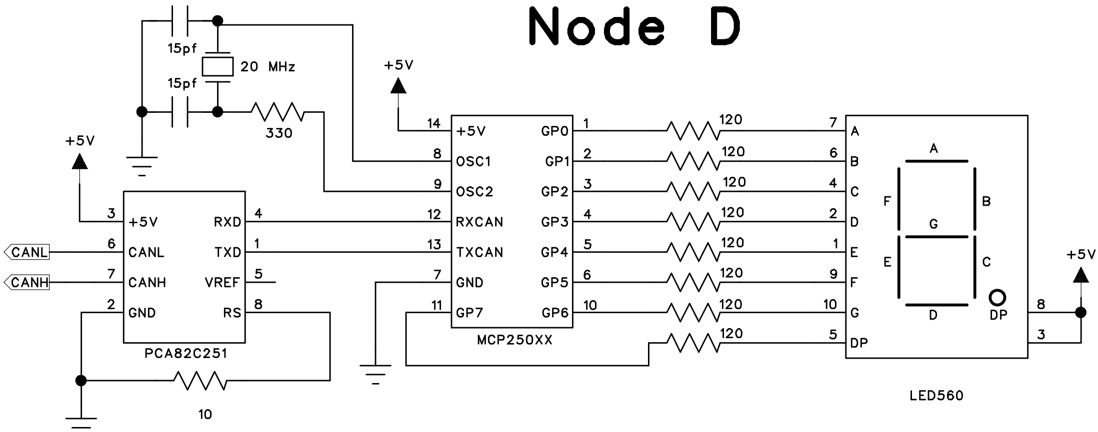


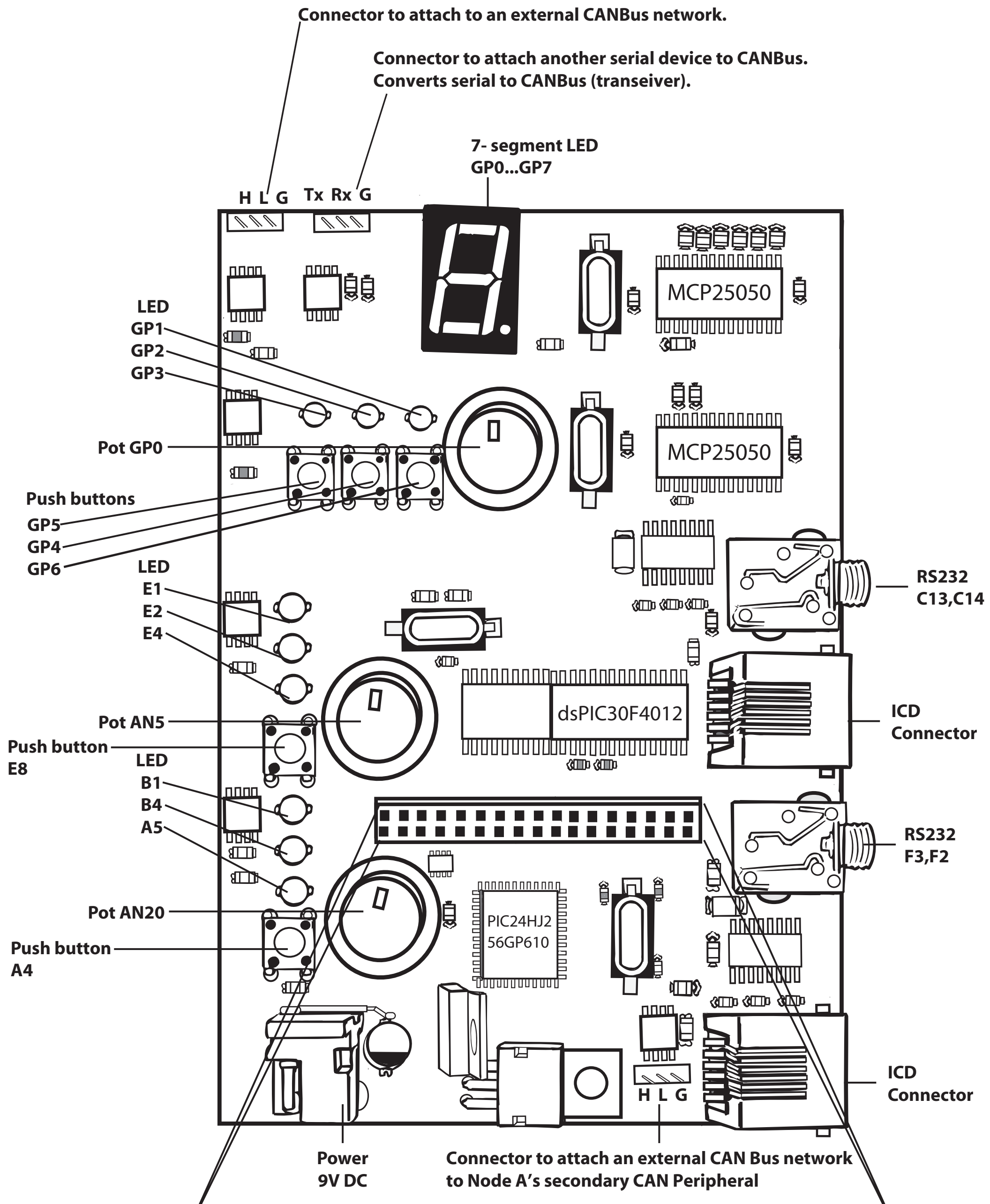
+5V
15pf
20 MHz
330
PCA82C251
RXD
TXD
CANL
CANH
VREF
GND
RS
10
MCP250XX
OSC1
OSC2
RXCAN
TXCAN
GP0
GP1
GP2
GP3
GP4
GP5
GP6
GP7
ANALOG SOURCE
RED
YELLOW
GREEN
10K

External



Node D





+3.3	E4	E2	G14	G12	E0	A6	D6	D4	D12	D2	D0	C14	D10	D8	A14	G
+3.3	E5	E3	G15	G13	E1	A7	D7	D5	D13	D3	D1	C13	D11	D9	A15	G